THE DEFINITIVE GUIDE TO EXPLORING FILE FORMATS

Mr.Mouse and WATTO

© 2004 XeNTaX.com & WATTO.org

Version 1.0, November 1st 2004

1. Table of Contents

1. Table of Contents 2
2. Introduction 4
What is a GRA? 5
What is a GRAF? 6
3. Tools 7
Hex Editors 7
Hex Workshop 8
4. Terms, Definitions, and Data Structures 10
Files 10
Bits 11
Bytes 12
16-bit (2-byte) numbers 13
32-bit (4-byte) numbers 14
64-bit (8-byte) numbers 15
Strings 16
Hexadecimal Numbering 17
Signed and Unsigned Numbers 18
Big-Endian and Little-Endian 19
File Offsets 20
5. Archive Patterns 21
Directory Archives 22
Tree Archives 23
Chunked Archives 26
Split Chunk Archives 27
External Directory Archives 28
6. Checking Your Results 29
Common Types Of Fields 29
Validating Your Fields 30
Padding 31
Filename Patterns 32
7. Encryption and Compression 33
7.1 The basics 33
XOR 34
NOT 35
SHL 35
SHR 35
7. 2 Encryption 36
Painkiller Encryption 37
Compression 44
8. Worked Examples 47
Quake *.PAK 47
9. Appendix 53
A: Binary → Byte Number Table 53
B. American Standard Code for Information Interchange (ASCII) Table 57
C. Format of some common Game Archives 59
D: Useful References 63

© 2004 Mr.Mouse and WATTO

E: Common File Format Tags 64
10. Legal Information 65

2. Introduction

Computer games are vast and many, however most computer games have something in common – they need a place to store all their important files like images, movies, and sounds. To do this, computer game developers typically store their data into a big archive file.

There are many reasons for storing all your data files in one big archive, some reasons include reducing the number of files on a CD, hiding the data files to stop people hacking the game, and so that all data files can be accessed using a single data stream.

However, the bad news for gamers is that there are almost as many different archives as there are different computer games – every game developer creates their own archive formats, and they even change their formats between games or departments in the company.

This brings us to the focus of the tutorial – how to explore the archives and grab the files from within them. This tutorial will attempt to make it easy for anyone to explore a new format, with the aim of promoting game modifications and enhancements by the community.

In the following pages, we will discuss the terms Game Resource Archives (*GRAs*) and Game Resource Archive Formats (*GRAFs*), common data types, and other definitions. From there, we will explain the fundamentals of cracking a file format, including the tools you use, and the patterns to look out for.

Thanks for reading our guide; we wish you the best of luck in your exploration☺.

© 2004 Mr.Mouse and WATTO

What is a GRA?

We should start this off by defining the definition of an archive. An archive is a file that usually stores many small individual data files inside it. The most well-known archive format that you will be familiar with is *.zip archive, which you should recognise as a way to package many files together into a single file.

A GRA (*Game Resource Archive*) defines an archive that is used by a specific *game,* and contains resource or data files that are used within the game, such as images, sounds, scripts, and text.

© 2004 Mr.Mouse and WATTO

What is a GRAF?

GRAF (*Game Resource Archive Format*) describes the way a game archive is constructed, and in particular, how and where the files are found within the archive. These formats usually differ for each individual game, however occasionally a game developer will stick with a particular format for a few games of the same vintage.

Programmers usually define their GRAFs according to the needs and structure of the game itself, i.e. the game engine. A common principle, however, is to use a format that can be quickly and easily be opened by the game, even though the files in the archive may be different types. For example, different levels in a game need different sounds and textures, so often the sounds and images for each level are packaged into the same archive, even though clearly images and sounds are handled differently.

In addition, the actual resources used in a game change frequently as the game is being developed. To make it easy for the game to adapt to the changes, the archive format is structured to be 'universal' in their approach of storing files. In simpler terms, the archive needs to provide the game engine with a common and recognisable pattern of resource files, and a list of the files that the GRA archive contains.

© 2004 Mr.Mouse and WATTO

3. Tools

Hex Editors

Hex editors are the standard class of program that you use if you wish to crack the formats of a newly encountered game.

The generic hex editor displays the contents of any file in a similar way that a piece of text in a word processor would be shown to you: from left to right, line by line, all the way down to the last character. However a hex editor differs from a word processor in that it shows the file as hexadecimal numbers rather than letters.

There are many hex editors available for free download over the Internet, however the one that we can fully recommend is Hex Workshop from Breakpoint Software. Hex Workshop will be the editor used in all the screenshots, and the one used in the examples. Hex Workshop includes some handy little functions, which is partly the reason why we recommend it, such as an easy to use hexadecimal calculator, lists of all data types that are at the current location of your cursor, bookmarking, and colour mapping.

© 2004 Mr.Mouse and WATTO

Hex Workshop

Examine Figure 1 carefully, as it shows the hex editor that will be used throughout this guide, Hex Workshop. After installation you can either start it and open any file you want using the File menu, alternatively (and this you will do most of the time) you can select to open files from the context menu in Windows Explorer. To do that, right click on any file in Explorer and select "Hex edit with Hex Workshop".
Once you have opened a file, you will be presented with a similar view of the file content as depicted in Figure 1. You can examine the file's hexadecimal interpretation (A), or ASCII interpretation of the same bytes (B). The table to the far left shows the offsets of the lines shown. As you can see, we have just opened one of Doom 3's PK4 files. We will later see that these are actually ZIP files. For now, you can see the file starts with the characters "PK". In this case, it is actually an ID tag that is used in all ZIP archives. PK stands for Phillip Katz, the author of the ZIP compression technique, who died at the age of 37 due to fatal effects of alcoholism in April 2000. The current cursor position is at offset 18 (see figure) and the Data Interpreter (C) shows the relative value at this file position. You can see it shows different data types (see Chapter 4) ranging from bytes to strings to binary values using the byte sequence starting at this offset. In the Figure, we have colour mapped and bookmarked (D) some areas of our interest. You can select any range of bytes in the file and ***bookmark*** or ***colour map*** it. Simply drag the cursor along your area of interest holding the left mouse button and right click the highlighted area to show the context menu from where you select the options. When you bookmark it, you can choose how the bookmark should be interpreted (*value*), and give a *description*. The bookmarks will be shown with their *offset* in the file and the size in bytes (*length*). This is a handy feature as you can click on any bookmark to quickly switch to that offset. When you leave Hex Workshop it will ask you if you want to save your colour map or bookmarks. Later, you can load these regardless of the file you have opened. Thus, if you have solved the puzzle of a GRAF, you can apply the bookmarks and colour mapping to other files that you expect to have the same format.

Hex Workshop has another handy ***GoTo*** method, which you too can access from the context menu. There are a number of options. With the basic GoTo option you can type in an address (offset) in the file you wish to go to. Two more powerful options are accessed by selecting a range of bytes (preferably one that you expect to represent an offset value) and then right clicking it. You can then either GoTo the value of your selected range or GoTo that the address of your value plus the current offset of your variable.
For instance, suppose you expect a 4-byte value (i.e. 32-bit) at offset 4 to represent the offset of a resource in a GRA. You can then opt to select the 4 bytes from this position (i.e. byte 4, 5, 6, and 7) and jump to the location in the file this 32-bit number points to. On the other hand, it might be that your file has an ID tag (GRAIS) of 4 bytes (i.e. bytes 0-3, see also offsets) and the GRAF does not count the 4 bytes of this tag when referring to offsets of resources. Thus the number you are labelling as an offset may in fact be a *relative offset*, meaning that it is relative to some other offset (in this case the offset where the GRAIS ends). You can easily jump to this relative location

with Hex Workshop, as the other GoTo option you are offered is to jump to the location your value depicts + the offset of the currently selected area. You can see that this program can come in very handy indeed if you are looking to unravel the puzzle of new archive formats.

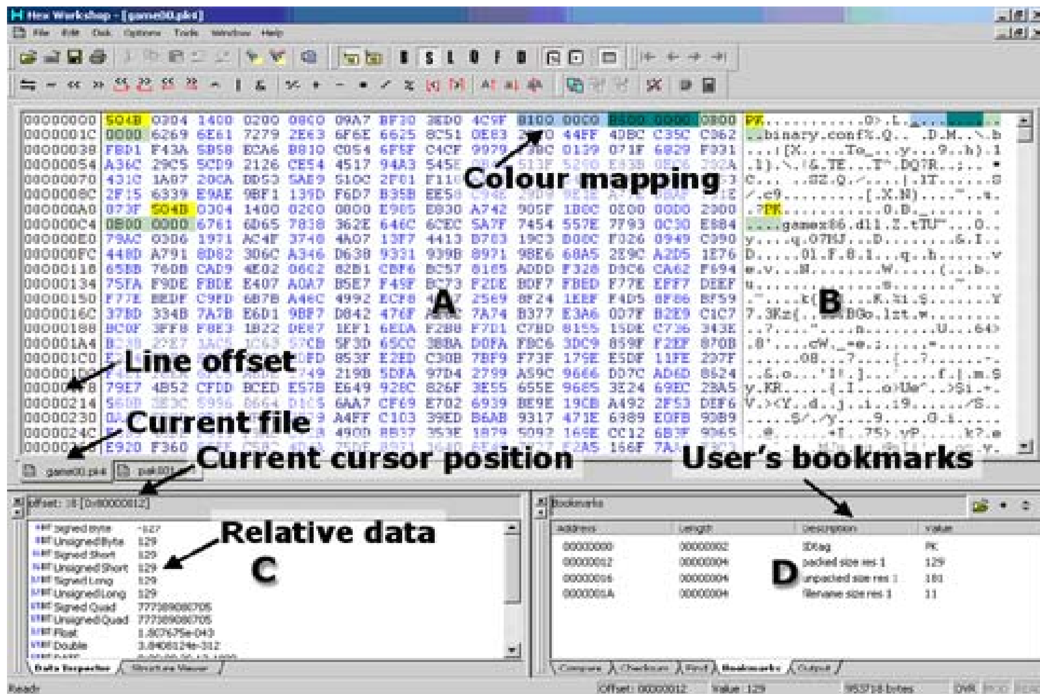

Figure 1. Layout of Hex Workshop. A. Hexadecimal representation of file content in sequential byte order. B. ASCII interpretation of file content. C. Interpretation of data at current cursor position. D. User window showing bookmarked areas in the file and the description given to these areas by the user.

4. Terms, Definitions, and Data Structures

To understand the patterns and construction or archives, we must first introduce the concept of data structures.

Files

Files are typically dealt with as a series of bytes stored one after the other, and when combined form a representation of a piece of data. If you have a file that is 12 bytes in size, it indicates that the file contains 12 single bytes that can be read in a certain way to represent a value. File sizes start off at the preliminary byte, followed by kilobytes (KB, 1024 bytes) and megabytes (MB) that are generic terms representing thousands and millions of bytes respectively.

© 2004 Mr.Mouse and WATTO

Bits

When we talk about the basic structure of a file, we typically think in terms of bytes. However, the actual underlying file structure, at its absolute simplest, is a sequence of bits or binary values, but we usually don't deal with this level of representation too often because binary values don't represent anything meaningful in their singular state – they need to be grouped into sets of 8 bits to become a valid representation of data.

A bit, or binary value, is the language of a computer, and thus the underlying structure of everything readable by a computer. A bit only has 2 possible values – 1 or 0 – thus it is obvious why they are limited in what they represent. However, when we take groups of bits and join them together, their values couple together into larger numbers that hold greater value.

The number of grouped bits needed to represent basic data is 8 bits, more commonly known as a byte. A byte can hold any value between 0 and 255, much more than the 2 possible combinations available to a bit.

So how do coupled bits represent a larger numerical value such as that of a byte? This is achieved quite easily by referring to each of the 8 bits as an increasing power of 2. If we take a look at a single bit, we can think of it as having either the value $1x2^0$ or $0x2^0$ – thus giving us the values 1 or 0. If we add a bit to the left, the power of the new bit is either $1x2^1$ or $0x2^1$ – either 2 or 0. By adding the values of these 2 bits together, you should be able to see that all possible combinations will give us the values 0, 1, 2, or 3. This is shown in the table below.

Bit 1 (2^1)	Bit 0 (2^0)	Value
0	0	0 ($0x2^1 + 0x2^0$)
0	1	1 ($0x2^1 + 1x2^0$)
1	0	2 ($1x2^1 + 0x2^0$)
1	1	3 ($1x2^1 + 1x2^0$)

If we continue this pattern for the remaining 6 bits (to make a total of 8 bits ie 1 byte) then we can provide powers up to 2^7, and thus if all 8 bits had the value 1 and we added them together, we would end up with the number 255. Appendix A shows all possible values of a byte, and the 8 bits used to create the value.

© 2004 Mr.Mouse and WATTO

Bytes

As described above, a byte is composed of 8 bits, and can thus contain a value between 0 and 255. When you open a file in a hex editor, each value you see represents a single byte value. You will probably not see numbers between 0 and 255 representing the bytes in a file; rather you will most probably see hexadecimal representations of the values, ranging from 00 to FF. For an explanation of hexadecimal numbering, see the Hexadecimal Numbering section below.

© 2004 Mr.Mouse and WATTO

16-bit (2-byte) numbers

From this point forward, we need to be careful what we call each of these data types. Why? Because each programming language uses different names for the same values.

A 16-bit value is commonly known in older programming languages (C++ or earlier) as a word or an Integer. Newer programming languages, such as Java and the .NET languages, call it a Short.

A 16-bit number is just as the name suggests, a number created by 16 bits in a row. To determine the value of the 16-bit number, we follow the same process as when we wanted to get the value of a byte.

Each of the 16 bits that make up the 16-bit number represents a power of 2 – the leftmost bit represents 2^{15} and the rightmost bit 2^0. Just as with bytes, we just go through each bit and calculate the *bitvalue x power*.

An example – lets say we have the following 16 bits…
101111000001100

Working from left to right, we get the value…
$1x2^{15}+0x2^{14}+1x2^{13}+1x2^{12}+1x2^{11}+1x2^{10}+0x2^9+\ldots+0x2^1+0x2^0$

If you work this out, you should end up with the number 24076.

If all 16 bits had the value 1, you would end up with the number 65535 – therefore the value of a 16-bit number ranges between 0 and 65535.

© 2004 Mr.Mouse and WATTO

32-bit (4-byte) numbers

If you have been following us so far, you should be able to see just how you would calculate the value of a 32-bit number. A 32-bit number uses 32 bits to represent it, and therefore has a value between 0 and 4294967295. This is a very large range, and thus is the reason why most values used in archives are stored in groups of 32-bits.

A 32-bit number in older programming languages is known as a dword or a Long. In newer languages, this is known as an Integer.

© 2004 Mr.Mouse and WATTO

64-bit (8-byte) numbers

As with the previous example, this is a number represented by 64 bits, and thus can contain any value between 0 and a massive 18446744073709551615. However, they can also represent floating-point numbers. These range in value from -1.79769313486232E308 to -4.94065645841247E-324 for negative values and from 4.94065645841247E-324 to 1.79769313486232E308 for positive values. These numbers are very rarely used in GRAs, but will be becoming more popular as things like 64-bit processors take off. Programming languages can refer to these as Longs, Doubles and even Floats.

© 2004 Mr.Mouse and WATTO

Strings

One of the most common tasks performed on a computer is word processing, so naturally we need some way of representing text in a document. A piece of text in a document is called a String, which more formally means a sequence of characters.

Although there are many languages in the world, and Spanish being the most spoken Latin based language, in Informatics the first Latin language used in the Western world is English. The English script consists of 52 letters (upper and lower case), 10 numbers, and about 30 symbols. Seeing as though this adds up to about 92, it seems quite logical that we can represent text as binary values in a byte (remembering a byte supports up to 255 different numbers). This is exactly what happens when you open a text document in a word processor – the word processor reads the bytes of the file and represents each byte value as a character.

For example, when the word processor reads a byte with value 65, it displays the letter ‘A’. The byte value 100 represents the letter ‘d’. You can then see that each byte value between 0 and 255 represents a letter, number, or symbol in the English alphabet. It is for this reason that if you open a non-text file in a word processor, you will see a lot of different letters and symbols – the word processor is displaying each byte as if it is a letter – because it simply doesn’t know that it isn’t a text file.

A character, therefore, is an 8-bit number that is represented as a readable English symbol. A group of characters written in sequence is known as a String, but only if it is intended to be read as English – if you see a group of English letters in a file that doesn’t make any sense, it may not be a String. Caution should be taken when discarding a string of characters. Nowadays many different countries sell mainstream programs, and more often than not you will find pieces of text that are in a different language. What could seem to the English speaking as gibberish might clearly be another language. For instance, these characters may represent oriental characters that will be translated into the right script if you have the software support. Japanese software is well-known and it is evident that they use their own language to describe files or events in their archives. Likewise, many titles come from Europe, so the strings could be in Russian, Finnish, German, Hungarian and so forth.

© 2004 Mr.Mouse and WATTO

Hexadecimal Numbering

Hexadecimal numbering represents a byte value in a convenient and compact way, and is the most common form of numbering used in a hex editor.

Recall that a byte can contain a value between 0 and 256. To write this number in hexadecimal, we split the number into 2 values, each representing a power of 16, much like binary numbers represent powers of 2.

Because we are talking about powers greater than 10, we need to add in some additional symbols to represent the numbers 10 through 15 with a single character. To do this, we use the letters A through F to represent the numbers 10 through 15. So, the letter C in hexadecimal represents the number 12.

Now how do we write a number in power 16? As mentioned earlier, the byte value is split up into 2, with the first number representing 16^1 and the second number representing 16^0. You may notice that this is similar to the way bits are joined together to form larger numbers.

The second number of the pair can take any value between 0 and 16 (labelled 0 through F), where the value represents *number x* 16^0. So, if the second number was 6, it would represent the number $6x16^0$ –the value 6. If the second number was B, it would similarly represent the value $11x16^0$ – the value 11.

The first number of the pair represents the value *number x* 16^1. So, if the first number was 2, it would represent the value $2x16^1$ – the value 32.

Let's look at a full example now. If we are given the hexadecimal value 1F, what does it represent? The number 1 means $1x16^1$ and the F means $15x16^0$. Added together, we get 16+15, the value 31. Similarly, the hexadecimal number E3 represents $14x16^1+3x16^0$, the number 227.

It should be clear to you now that we can represent a byte (values 0 through 255) in the hexadecimal number system using the values 00 through FF. Here we print hexadecimals as *&h* <value>.

© 2004 Mr.Mouse and WATTO

Signed and Unsigned Numbers

So by now you are clear as to how numbers are stored in files, and even how strings are stored, but what about negative numbers? Luckily, negative numbers are really easy.

There are only 2 possible classes of numbers used here – either positive or negative. This maps perfectly with a single bit of value 0 or 1 respectively.

Rather than add an extra bit to a number, we take the bit with the highest value and interpret it as a positive or negative sign rather than contributing to the number. For example, in an 8-bit number, you would count all the bits from 2^0 to 2^6, and the value of the 2^7 bit will determine whether the value it positive or negative. You should note that because the highest bit is being used for another purpose, it cannot be used as part of the number itself. This effectively cuts the possible values of the number in half. In our example, you would normally be able to have any value between 0 and 255; however with the negative bit we now have numbers between -127 and 127. Also note that 0 and -0 both indicate the number 0.

Here we need to introduce a way of knowing whether a number will be positive-only, or a positive/negative number. We therefore use the term *signed* to indicate that the highest bit is used as a sign, or the term *unsigned* indicating the number is positive. Therefore, if you are told a 16-bit number is unsigned, you will know the number ranges between 0 and 65535. However, if it was a signed 16-bit number, it would range between –32767 and 32767.

Note that for archives, it is very rare that you would need to use signed numbers. Unless mentioned, you should assume all numbers used in archives and in this document are unsigned.

© 2004 Mr.Mouse and WATTO

Big-Endian and Little-Endian

If you paid close attention, you would have noticed that whenever we calculate a number, the bit with the highest value was always on the left, and the lowest value on the right. This is regarded as almost a standard today amongst PC users, however some files, programs, and computer systems decided it was better to read it the other way around (ie right-to-left instead of left-to-right). So once again, we need to define some terms so that people know what order we are talking about.

Little-Endian order is the one we will be using in this document, and unless stated specifically you should assume that Little-Endian order is used in any file. The alternate is Big-Endian ordering.

So let’s see an example. Take the following stream or 8 bits

10001110

If you have been following the document so far, you would quickly calculate the value of this 8-bit number as being

$$1x2^7 + 0x2^6 + \ldots + 1x2^1 + 0x2^0 = 142$$

This is an example of Little-Endian ordering. However, in Big-Endian ordering we need to read the number in the opposite direction

$$1x2^0 + 0x2^1 + \ldots + 1x2^6 + 0x2^7 = 113$$

It is always important to read the numbers in the correct order, otherwise you will end up with numbers that are meaningless and incorrect. As mentioned, if you don’t know which order to use, assume Little-Endian ordering. We will use Little-Endian order for all examples in this document.

© 2004 Mr.Mouse and WATTO

File Offsets

One of the most fundamentals of format exploring is the concept of file offsets. A file offset is the position of a certain piece of data in a file, measured from the first byte of the file. However, as with most computer programming, we start our number counts at 0, not at 1. Therefore, if we are at the very beginning of the file, before we read anything, we are at offset 0. After we read 1 byte, we are at offset 1. Read another 6 bytes and we are at offset 7.

If the concept is a little hard to grasp, think of an offset as being a bar that divides a file up byte-by-byte. If we are at the beginning of a file, offset 0, we have a bar right at the beginning before the first byte

I011000101101100000101110

If we are at offset 3, we place the bar after the 3rd byte of the file, and before the 4th byte

011I000101101100000101110

Similarly, offset 16 places the bar after byte 16, and before byte 17

0110001011011000I001011110

© 2004 Mr.Mouse and WATTO

5. Archive Patterns

There are literally thousands of different archives out there, however most archives will conform to one of several basic formats. Here we present the basic archive patterns so you can help pick the basic archive type, and once you know that you can make progress faster.

Note the graphical samples presented here list some fields. These fields are not fixed, and indeed may be totally different to your archive. You should use the samples supplied as a guide to the overall structure only, not as a specific guide for a format.

Also note that in many cases, but not all, the first few bytes indicate the name of the format the current archive has. For instance, in the example in Figure 1, we opened a PK4 archive from Doom 3, and it's first 2 bytes gave away the format the archive has: PKZIP (remember they read 'PK'). We will call them game resource archive *identity* *strings* (GRAIS). The length of these strings may vary per archive investigated, and may also be completely absent.

© 2004 Mr.Mouse and WATTO

Directory Archives

Directory archives are by far the most common archive structure in use today. Directory archives work by storing a directory somewhere in the archive that outlines the offset of each file in the archive.

Usually the directory is presented at the beginning of the archive (called a *header*), as in the example below, however occasionally the directory will be placed elsewhere in the archive (typically at the end and called a *tail*). If the directory is not at the start, there will be a field somewhere that gives you the position of the archive – after all the game itself needs to know where the directory is. The directory offset is usually a 4-byte field either somewhere in the archive header, or at the very end of the archive.

Here is a sample graphic representation of the archive

Archive Header
4 - GRAIS (String)
4 - Number of Files
Directory
    File Entry 1
    4 - File Offset
    4 - File Size
    X - Filename
    File Entry 2
    4 - File Offset
    4 - File Size
    X - Filename
    …
    File Entry n
    4 - File Offset
    4 - File Size
    X - Filename
File Data
    File Data 1
    File Data 2
    …
    File Data n

© 2004 Mr.Mouse and WATTO

Tree Archives

Tree archives are more complicated archives that attempt to store a complete directory tree structure inside a file. This is usually done by defining a group of directory entries, where each directory entry points to another directory entry, and so forth until you reach the folder that contains the files, which then progresses as in the directory archive.

Here is a sample graphic representation of the archive

Archive Header
4 - GRAIS (String)
4 - Number of Directories at Root
4 - Number of Files
Directory Entries
    Directory Entry 1
    X - Filename
    4 - Subdirectory Offset
    4 - Number of Files in Directory
    4 - Number of Subdirectories in Directory
    Directory Entry 2
    X - Filename
    4 - Subdirectory Offset
    4 - Number of Files in Directory
    4 - Number of Subdirectories in Directory
    …
    Directory Entry n
    X - Filename
    4 - Subdirectory Offset
    4 - Number of Files in Directory
    4 - Number of Subdirectories in Directory
File Entries
    File Entry 1
    4 - File Offset
    4 - File Size
    X - Filename
    File Entry 2
    4 - File Offset
    4 - File Size
    X - Filename
    …
    File Entry n
    4 - File Offset
    4 - File Size
    X - Filename
File Data
    File Data 1
    File Data 2
    …
    File Data n

© 2004 Mr.Mouse and WATTO

As these archives are quite difficult to explain, I will provide an example here. Let's pretend we have 3 files in the directories specified below

\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***
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

© 2004 Mr.Mouse and WATTO

File Data 1
File Data 2
File Data 3

Let's walk through the reading of this file.

First we read the *archive header* and see that there is only 1 *directory at the root*. This lets us know that we now need to read a single directory entry.

We read directory entry 1, called ***temp***, and are told there are 2 subdirectories, and the directory entries for these subdirectories start at a certain offset.

So we skip to the offset of the subdirectories. For each of the 2 subdirectories, we need to read a directory entry. The first directory entry read is called ***sounds*** and there are 2 files in it. The second entry is ***images*** and there is a single subdirectory in it.

So we jump to the ***sounds*** *offset* and read 2 file entries, namely the file entries 1 and 2. After we have read these, we jump back to the ***images*** *offset* and read 1 directory entry, called ***temp***, which has 1 file in it.

We jump forward into the ***temp*** *offset* and read the 1 file entry.

Using this method, we can build up a complex directory tree. This type of archive is usually slightly smaller in size than the plain directory archive, however the compromise is that it takes longer to read because you are jumping all over the place. For this reason, and the fact that it is a very complex structure, only a rare few games use this type of structure.

© 2004 Mr.Mouse and WATTO

Chunked Archives

Chunked archives store their files one after the other, with each file containing a header giving information such as the size and type of the file. These archives, probably the simplest of all archives, are examined by reading the header of the file, skipping the file data, then repeating again for the remaining files until you reach the end of the archive.

These archives are based on the Electronic Arts IFF85 standard that is used for many files including WAV audio and AVI video.

Here is a sample graphic representation of the archive

Archive Header
4 - GRAIS (String)
4 - Archive Size
    File Header 1
    4 - File Type (String)
    4 - File Size
    File Data 1
    File Header 2
    4 - File Type (String)
    4 - File Size
    File Data 2
    …
    File Header n
    4 - File Type (String)
    4 - File Size
    File Data n

The *archive header* typically contains a 4-byte GRAIS, and a 4-byte field indicating the size of the archive.

The *file header* typically contains a 4-byte type tag, and a 4-byte file size field, however other fields may be included such as filenames and file IDs.

© 2004 Mr.Mouse and WATTO

Split Chunk Archives

These archives are similar to the chunked archive. However, each file is also split up into chunks. Each file chunk is the same size, which allows efficient use of buffers when reading the file.

Here is a sample graphic representation of the archive

```
Archive Header
4 - GRAIS (String)
4 - Archive Size
    File Header 1
    4 - File Type (String)
    4 - File Size
    4 - Number of Chunks
    4 - Chunk Size
        File Chunk 1
        File Chunk 2
        …
        File Chunk n
    File Header 2
    4 - File Type (String)
    4 - File Size
    4 - Number of Chunks
    4 - Chunk Size
        File Chunk 1
        File Chunk 2
        …
        File Chunk n
    …
    File Header n
    4 - File Type (String)
    4 - File Size
    4 - Number of Chunks
    4 - Chunk Size
        File Chunk 1
        File Chunk 2
        …
        File Chunk n
```

Note that as each chunk is a fixed size, the *chunk size* field need only be specified once for each file. This field may even be specified in the archive header if the chunk size is fixed for all files.

© 2004 Mr.Mouse and WATTO

External Directory Archives

External directory archives have the same structure as the directory or tree archive. However, the directory data and the file data are stored in 2 separate files. Naturally, the file that contains the file data is very large, and the directory file very small.

Here is a sample graphic representation of the archive, where the directory archive format is used

file.dir
Archive Header
4 - GRAIS (String)
Directory
    File Entry 1
    4 - File Offset
    4 - File Size
    X - Filename
    File Entry 2
    4 - File Offset
    4 - File Size
    X - Filename
    …
    File Entry n
    4 - File Offset
    4 - File Size
    X - Filename

file.arc
File Data
    File Data 1
    File Data 2
    …
    File Data n

Note that the 2 files both have the same name, but different extensions. Also note that the extensions are not fixed to the extensions given in the example, they can be anything so long as the 2 files have different extensions.

© 2004 Mr.Mouse and WATTO

6. Checking Your Results

Common Types of Fields

Archives can literally contain fields for just about any purpose; however it helps to know some of the more common fields as you then know what to look for.

The following fields are very common in archives, so there is good probability that you will run into at least one of these fields.
- File Size
- File Offset
- Number Of Files
- GRAIS

The following fields occur in some archives, but at significantly less probability compared to those listed above.
- 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

© 2004 Mr.Mouse and WATTO

Validating Your Fields

When you think you know what a field means, it is important to validate your findings. One of the simplest to check are the *file offset* and *file size* fields. If you are presented with these two fields, simply add the first file offset to the first file size and see that it matches the second file offset. Repeat this a few times, and you can pretty much guarantee that those 2 fields are correct.

Another field that is easy to validate is the *file offset* field, if you choose a good archive. All you need to do is go to the offset for each file and see if it points you to a known file header. For example, if you open a sound archive then a good header to look for is RIFF as it indicates a *.wav sound file. Similarly, if opening a texture archive, look for common image headers such as BM (*.bmp), GIF (*.gif) and JFIF (*.jpg). So if you pick the right archive, you can see whether the file offset field is correct.

If you think that an archive compresses its files, and you have found the *file size* field, try looking for a *decompressed file size* field for each entry – simply look for a field that is always a little larger than the size for the file.

If you locate a directory in the archive, try to find a constant file entry size if possible. For example, if each file entry contains a file size field and a file offset field, then each file entry has a size of 8 bytes. Once you know this, work out the size of the directory by finding the offset to the end of the directory and subtracting the offset to the start of the directory. When you divide the directory size by the file entry size, you will be able to find out the number of files in the archive. This number may be stored in the archive somewhere, usually at the start of the archive, so look out for it. Otherwise, you can just perform the same calculation as you did above when you go to implement this format in a program.

© 2004 Mr.Mouse and WATTO

Padding

So you think you have determined the fields and their data type, but for some reason you can read a few of the entries and it just starts getting it all wrong. This is most probably due to padding, the technique of adding null bytes to the archive as a way of expanding the data to a fixed length.

For example, some archive types allow you to have a filename which is of an arbitrary length, such as where the filename is stored followed by a null byte, then the directory continues. As you can see, each entry will have a different length depending on the length of the filename. This may not seem like a problem, but when using buffers to read a file the buffer may become mis-aligned. This is where padding comes in to play.

Entries in a directory are commonly padded to multiples of 4 bytes. If the filename length is not a multiple of 4 bytes, a number of null bytes is added to the end of the filename to bring it to the correct size. These bytes are to be ignored; they are simply there for better reading of the file.

Let's say you have a filename of length 7. The next multiple of 4 bytes that occurs is the number 8 (4x2), so only 1 null byte needs to be added. Similarly a filename of length 13 will need to be padded to a length of 16 (4x4) so we need to add 3 null bytes.

Some archives also like to pad out their files to multiples of a number, typically multiples of 2048 bytes.

© 2004 Mr.Mouse and WATTO

Filename Patterns

Filenames are a very important thing to find in an archive because it tells the public how to open and edit the file. Some archives store filenames, however many do not because they take up a lot of space to store, and normally the game does not use the names. Archives that do store filenames have plenty of choices concerning how to do just that.

If filenames are not used by the game, sometimes the filenames are stored in their own directory, thus allowing the archive to be read quickly and efficiently by the game (by simply skipping the filename directory completely). For this structure, you are usually provided with the filename directory offset somewhere in the archive (typically at the start of the archive), so you know where to find the filenames. The filenames are usually stored in the archive one after the other, in the same order as the files are stored in the archive. Each filename is normally separated by a single null byte.

If the filenames are not stored in a separate directory, you may be lucky to find the filenames stored with each file entry in the directory. Wherever the filename is stored, they fit into one of a few different formats.

The most common format is the storage of the filename, followed by a single null byte.

A slight variation on this format is that for buffering, the filename must fit a multiple of a certain size. For example, filenames are usually padded to a multiple of 4 bytes, where if the filename length is not a multiple of 4, null bytes are added to the end of the filename to make it up to the correct size. More information on this is found in the Padding section.

Another possible format is that each filename is stored in a fixed number of bytes. If the filename is too short, the filename is expanded to the correct size by adding null bytes to the end of the filename. If the filename is too long, it is just cut off at the correct size.

On really good archives, a field just before the filename will actually tell you the length of the filename. This field is usually either a 4-byte field, or just a single byte. This is really handy because then you know exactly how long the filename is. The filename may or may not have a null byte following it – if it does then the null byte is normally included in the filename length field.

You may see that the overwhelming number of filename formats rely on the filename being followed by one or more null bytes. For this reason, we typically call a filename a *null-terminated string*, ie a string that continues until you reach a null character. We should note here that, for those of you that don’t know, a null byte is a byte of value 0. Also, a slight variation on these formats, some archives will use the character 32 (the space character) instead of the null byte.

7. Encryption and Compression

7.1 The basics

In many cases, programmers use different techniques to either protect their resources contained in GRAs, by *encrypting* them, or limit the size of the archive by *compressing* the resources before creation of the archive.
There are many different methods to encrypt, which makes it hard to figure out. Almost any programmer can come up with his own routine that will be very difficult to crack just looking at the archive. We will still try to shed some light on this subject, as encrypted and compressed archives will become more commonplace in the future. There are many tools you can utilize to crack the codes, besides the obligatory hex editor. For instance, try to get a disassembler, such as WinDASM, and use it to open the game's executable with. With the mentioned tool, you can easily show every text string that the game uses (such as archive filenames, but also resource names!). You will see later why this can come in very handy! Not only that, but if you are a sophisticated programmer, that knows how to interpret *assembly* code ("machinelanguage"), you can try to backtrack whatever it was the executable did to decrypt the code; more on that later as well.

Paramount to understanding encryption is knowledge of *bitwise operations.* Bitwise operations can be regarded as simple logical steps where two bytes undergo a bit transformation into a resulting byte. The primary operations are And, Or and XOR (exclusive Or) What all of these do is change the bits in the first byte depending on the bits in the second byte. Other operations include NOT, SHL (shift-left) and SHR (shift-right) that act on a single byte.

AND

The AND operation sets a bit to true *only if both operators (read BIT) are true* like this:

0 AND 1 = 0
1 AND 0 = 0
0 AND 0 = 0
1 AND 1 = 1

Example: 12 AND 123

00001100 (12)
01111011 (123)   (AND)
___
00001000 (8)

OR

The OR operation sets a bit to 0 only if both operators are 0.

0 OR 0 = 0
1 OR 0 = 1
0 OR 1 = 1
1 OR 1 = 1

Example: 12 OR 123

00001100 (12)
01111011 (123)
___
01111111 (127)

XOR

The Exclusive OR operation sets the resulting bit to 1 *only if one of both operators is true*.

0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0

Example: 12 XOR 123

00001100 (12)
01111011 (123)
___
01110111 (119)

© 2004 Mr.Mouse and WATTO

NOT

The NOT operation is not really like the previous, in that it only applies to one byte. What it does is invert the bits in the byte:

00001100 (12)

11110011 (243)

SHL

The shift-left operation shifts the bits in a byte to the left, discarding the left-most bit and completing the bits with a zero at the right end. Basically, one shift-left is the same as $n*2^1$ (with n being the original byte). Two shift-lefts is $n*2^2$, three $n*2^3$ etc. up to a maximum of 255 of course, after that bits are discarded.

00110011 (51)

01100110 (102)

SHR

The shift-right operation shifts the bits in a byte to the right, discarding the right-most bit and supplementing from the left with a 0 bit. This is the same as a rough division by two, rounded to the lowest value in case of discarding a bit.

00110011 (51)

00011001 (25)

What can you do with these operations? Well, there are a lot of useful things you can do, like super-fast divisions by two, color inverting, but also turning specific bits on and off. Hardware, such as graphic cards, encodes specific functionality in bits, not in bytes. For example, suppose a screen may have a resolution of 640x480 pixels, but we only have screen memory for 4 banks of 320x200 bytes. How can we let the graphic card know which pixel to show in what color. Our graphic card will agree to a resolution of 640x480, but only if we divide the bytes into two and do some bank switching. What we get is rows of 320 bytes that are each actually two pixel positions next to each other. Note that each pixel only has 4 bits to work with so the maximum color value is all 4 bits set (15). Thus, we have a resolution of 640x480x16. Okay, but suppose we want to set the 4th pixel on screen from the left at the top row to color 15? But we want to keep the 3rd pixel the way it is? We should address the 2nd byte in memory and only change the high part of the byte (the left 4 bits). Now, we can't just say, byte2 = 15, because that would set the right part of our byte and clear the fourth pixel! We saw above that we have a way of keeping certain bits in a byte as they were. We need to use the OR operation, setting the first 4 bits of our color byte to 0 and the top 4-bits to the color we want the fourth pixel to have. This is easy! Remember shift-left (SHL)? We set

© 2004 Mr.Mouse and WATTO

our color byte to 15, SHL it 4 times and then OR the 2nd byte in screen memory with the resulting color byte:

1. color byte = 00001111 (15)
2. SHLx4 = 11110000 (240)

There, now suppose the second byte in screen memory is 178, then we do 178 OR 240 to set our pixel of interest:

10110010 (178)	– note that the third pixel (the lower half) has color 0010 (2), and the fourth color 1011 (11).
11110000 (240) OR	
11110010 (242)	

We did it! We changed the fourth pixel into color 15 and kept the third pixel in color 2.

This was just one of many uses of the bitwise operations.
Here it is important to realize you can do some hefty encrypting with this as well, especially if you combine them to good effect. The XOR operation is used many times in encryption techniques, so if you're looking to decipher some archives, remember to include XOR in your way of thinking.

7. 2 Encryption

One can ask a very simple question: what is encryption? Basically, encryption is a way to mask the true nature of a script. Files are nothing more than scripts that use up to 256 unique characters (bytes!) to "write" a "story". Every one of you has probably done some word puzzles like anagrams. Anagrams are just another, be it simple, way of encrypting a word or sentence. However, true encryption requires people to be able to reverse the encryption in a logical manner. Usually, this is not the case with anagrams. There's no universal logic applied when people create anagrams. However, when programmers encrypt their files or parts of their archives (usually those parts that contain important information, such as resource names, offsets and sizes) they will want their game editor to be able to de-encrypt (decrypt) the archive. As said, there's not just one way to encrypt and that makes it impossible to cover this subject to complete satisfaction. However, as most encryption techniques are purely logical in nature, the smart investigator can come a long way in determining the rudiments of the technique or even crack the code as a whole. How to proceed?

First of all, you should be absolutely positive that a part of the archive you are looking at is indeed encrypted. There's no point in trying to decrypt a pile of bytes that were not encrypted in the first place. So, (1) *identify an encrypted block of bytes*. If you are sure, try to find more examples of files that use this supposed technique. In other words, if you have discovered a putative but encrypted GRA, (2) *try to find more* and use them to compare specific parts; many games have more than one archive, but most use only one way of archiving. If you are positive that you are dealing with an archive and you can

identify some directory structure, you can also use this structure to compare the way individual (encrypted) directories entries are shown in the one archive. (3) Use a good disassembler to open the game's executable to *search for resource name strings*. You might just find a lot of strings there. If you can identify a resource by name, and you have already figured out where in the archive this resource is saved, you have a good reference for your decrypting. Suppose you have stumbled on a GRAF that encrypts the names of the resources before saving this information in an archive. Upon examination of the executable you may have found a string "/textures/weapons/pistol.pcx". Suppose furthermore that you already extracted some of the textures as nameless pictures. If you recognize the pistol texture from the game, you then have something to work with while decrypting, as you can compare the encrypted string with the decrypted.

But how about the actual decrypting? Where to start?
As said, there's not just one method of encryption. One trick that will tell you exactly what is going on is to run the executable from a disassembler and **reverse engineer** the assembly code that takes care of the encryption. This requires substantial knowledge of assembly language, though, and may be a slow and largely unrewarding process. Nevertheless, if you can pinpoint the location of the encryption/decryption code this may help you understand parts of it anyway and may be very worthwhile in the long run. As your understanding of these assembly processes grow the more easily and speedily you will figure these things out!

Pen and paper ready!

Without the aide of reverse engineering it is still possible to find out how encryption works. Pen and paper are two very handy tools that can help you solve the puzzle. We will try to explain the process of unravelling the Painkiller .PAK string encryption technique.

Painkiller Encryption

When the developers of the game Painkiller first released a demo, fans quickly discovered that the game used adapted PKZip files to store the resources (so called .PAK archives). Apparently, the coders did not want fans to have access to the resources, because in the second demo and the first retail version, they used another, more difficult to hack method of storing resources. While they used straightforward properties such as resource size and resource offset variables, they encrypted the filename of the resources. In addition, they changed the method to compress the actual resources to Zlib compression. Quickly people had got the whole format explained, and could extract the resources, but for the encrypted filename. That proved the hardest puzzle to solve.

© 2004 Mr.Mouse and WATTO

In the archive “scripts.pak” the first few encrypted filenames strings are shown in ASCII like this:
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

The first thing you do is take a good look at what you see. Try to see what can be seen. That may sound cryptic, but it’s true.
In our case, what do we know? Well, we know that the strings we are examining represent some king of encrypted text. We also assume these strings are in fact filenames. Good. That gives us something to go on. In addition, you did not know this before, but we tell you now: the 32-bit (4-byte) value that precedes each string is exactly the size of the string in bytes, and thus it is probably save to assume that the characters from the encrypted string match in position the characters of the original string. This is very important. Because with this knowledge we can make other assumptions:

Filenames usually have the following structure:

Directory\directory\filename.extension
(the number of directories may vary)

The extensions of filenames are usually 3 bytes in length.
Like: “text.doc”, where the “doc” is the extension.

Right, now take a good look at the strings. You will notice that they all start with a byte somewhere in the &h60+ range, followed by characters in the &h40-&h60 range, followed by a single byte in the &h10+ range. Wait a minute! The standard filename structure as shown above starts with a capital character and is then followed by small characters, followed by a backslash (e.g. *Directory*\). You should know that non-letter characters mostly have an ASCII-value (decimal value) which is far from letter-characters. Thus, we recognize this pattern in the encryption example:
<*capital*><*small characters*><*backslash*>…etc. Without knowing the original value of the letter characters, we can assume that the encrypted values &h1C, &h1B and &h16 in the subsequent strings are in fact ‘\’, or &h5C! We keep in mind that this may also be a forward slash though, ‘/’, or &h2F.
Close examination of the other characters in the strings reveal that the last three characters of each string are all preceded by a byte value in the &h60+ range. In the standard filename structure the last three characters commonly represent the extension of the file, with the character before the extension always a ‘.’ or &h2E. So it’s save to assume the following structure of the encrypted strings:

© 2004 Mr.Mouse and WATTO

<*Capital, small characters*>’****’<*small characters*>”**.**”<*small characters*>

Or, in function:

<*Directory*>’\’<*filename*>’.’<*extension*’.

And applied to the example of encrypted string 1:

<cLHLCB>’\’<P[\XIM>’.’<m,)>

However, we still don’t know what, for instance, “cLHLCB” stands for. Apparently there’s some algorithm at work here. Remember that you have logical ways to alter existing bytes, using XOR, AND and others? Let’s consider the often-used XOR. Could we use XOR to create such a string? First of all, you’d need to be able to “seed” the XOR encryption: Every string has a first letter, in our case the capital character. To get a XORed value for this capital, you need another byte to XOR it with. This is your “seed” value. Next, you perform some trick on the XOR value, before you use it to XOR the second character in your string, then you perform the trick on the XOR value again to XOR the third character and so forth. The crucial XOR thing is: by XORing a value by the same value it was XORed with, you retrieve the value that was XORed. In our case, a XOR method much like the one described was assumed, also because some people used a disassembler to look at the encryption code, and came across some XOR statements. And that really fits nicely, because why go to the trouble of encrypting your filenames, if you won’t need to decrypt them.
Thus, there must be some algorithm at work here which can be used in reverse as well: by applying the algorithm to the original text you will get the encrypted text, by applying the same algorithm to the encrypted text you will get the original text.
XOR is one trick you can use to create just such an algorithm. We know that we have a ‘\’ (or ‘/’) and a ‘.’ in our strings. Let’s take the first string. We can find out what the value must have been that was used to XOR our ‘\’ or ‘.’ and resulted in the encrypted code for these characters &h1C and &h6D respectively. Simply XOR them with the original value!

Tip: In Hex Workshop, you can select any range of bytes and XOR it with a value of your choice. Click on the “^” button at the top toolbar to do so. In this case, you would select a byte of interest and select an 8-bit unsigned value to XOR it with. The outcome would replace the selected byte.

Thus &h1C XOR &h5C = &h40, &h6D XOR &2E = &h43.
But we kept in mind that the proposed backslash may also be a forward slash. That would give &h1C XOR &h2F = &h33.

Now, let’s consider this difference in XOR value used. The distance between the back- or forward slash and the dot is 8 bytes. The difference in XOR value used is either &h43 -&h40 = &h3 (3) in case a backslash was used, or it is &h43 -&h33 = &h10 (16) in case of a forward slash. Could it be as simple as adding two to the XOR value (hence 16/8 bytes distance = 2) for each

subsequent XOR you do? We can test this by starting from the forward slash (if the theory has a chance we are dealing with the forward slash after all) and XORing the string characters to the last character in the string. Thus, we will use &h33 for the forward slash, then use &h35, &h37, &h39, &h3B, &h3D, &h3F, &h41, &h43 (our dot!), &h45, &h47 and &h49 respectively to XOR the next characters.

.P[\XIM.m,) = /electro.ini

So, it is indeed like that. Now we do this for the whole string (i.e. start from the forward slash with the XOR value of &h33 and subtract 2 from this value, XOR the characters all the way to the first):

cLHLCB.P[\XIM.m,) = Decals/electro.ini

So the first string is decrypted! But we still do not know how the “seed” XOR was calculated. The first character in our original string is the letter D (&h44). The encrypted value was &h65, thus the XOR used was &h27 (39). This is our seed value for the XOR method of encrypting the string. But how was it calculated?

We should first check the other strings and see what seed value we obtain for those. The above-described method (pinpoint the encrypted forward slash, XOR it with &h2F, use the XOR value to XOR the next encrypted characters after subsequent additions of 2, and the previous encrypted characters after subsequent subtractions of 2) will give:

lOOO\A.[WVSJ/4j/&#_ = Decals/molotov.ini
iJRRYD.IR*&1/&'(a8=< = Decals/rockethole.ini

We find that the encryption of the second string was “seeded” with &h28 (40) and the third with &h2D (45). You should notice they are all somewhat in the same range.
We assume that the program that wishes to decrypt the strings (like the developers’ game editor for instance) can get its “seeds” based on variables from the archive, and this should be specific per string (or in effect per resource). Let’s see what other variables we have. When you look at the whole GRAF of the Painkiller .PAK files you will see resource offsets and sizes, as well as the size in bytes of the filename strings, among other stuff. Now compare the sizes of the strings. The first is &h12 (18), the second also &h12 (18) and the third is &h15 (21). These variables you find as 32-bit values saved just before the encrypted string. Compare the “seeds” we found for each string. The first was &h27 (39), the second &h28 (40) and the third &h2D (45). Notice how the first and second string are equal in size and their “seed” is only a difference of 1.
To start with, let’s propose that the encryption method uses the string size variables to calculate the “seed”. Strings 1 and 2 both have the same size, so in theory they should end up with the same “seed”. However, string 2 has a “seed” value of 1 higher than string 1. Well, it is the second string after all, so perhaps the method will take into consideration the position of the resource in

© 2004 Mr.Mouse and WATTO

the file (the first one is file 1, the second file 2 and so forth). When the seed would be calculated, the final value could be incremented with the position in the file. This way the difference between the seed value of string 1 and 2, having the same size value, would indeed be 1. This then implies that the calculated "seed" value of &h27 *less* the position in the file for the first string would actually be &h27-1=&h26 (38), for the second &h28-2 = &h26 (38) and the third &h2D-3=&h2A (42).
Assuming the above is correct, then how can we obtain the "seed" value of 38 for the first string? This is not easy, but we just try a number of ideas. The size variable for the first string is &h12 (18). If we do a shift left of this number (multiplication of 2) we get &h24 (36). This is rather close to 38, isn't it? How about the third string? A shift left of &h15 gives &h2A (42). Hmm, this is exactly the seed value of the third string (not adding the position in the file)! Thus, if we calculate it this way for the first and second string we get a value that is 2 lower than what is should be, and for the third the difference between our calculation and what it should be is 0. Perhaps we are mistaken, and the method is different? Well, we are trying to understand, and obviously we haven't cracked it completely. We will still stay on track though and keep the shift left as it is rather close to the actual "seed" value.
More information is needed at times like this, and it is advised to apply your proposed methods on many cases of whatever is encrypted. In our case, we must check more strings to map potential differences in shift left value of the size variable and the "seed" value. We will not show it here, but we'll present the number of possibilities that you will get if you do so. We find that our shift left of the size variable differs from the "seed" value in this range:

shift left (size) – seed value = {-3, -2, -1, 0, 1}.

So the maximum difference is 3 less than the "seed", or the other way, 1 more than the "seed". This does point to some kind of tabular method, although the range is not very symmetrical. More symmetrical would be a range of {-2, -1, 0, 1, 2}. Let's see if we can alter our method so we have values that differ in that more symmetrical range. What would be needed? Well, all range values would have to be incremented with 1, right? –3 + 1 = -2, -2 + 1 = -1 etc.
For us it means that our shift left value of the size variable is one off. Remember that the common operation *shift left* shift all bits one to the left, *discarding* the left-most bit. However, there are also adaptations of the shift left method. First, there's *rotate left* that will rotate the bits one to the left, and the left-most bit will be rotated to the right-most bit. Thus, if the left-most bit is set it will transfer to the right-most bit. If that is the case it will result in a decimal calculation of value * 2 + 1. And that is what we need as well. However, if the left-most bit is not set, the right-most bit will simply be 0. In our case of the size variables, the left-most bit is never set, so the rotate left method will not get us where we need to be. The second adaptation of the shift left method is *inclusive shift left*. This will shift the bits 1 to the left, but now set the right-most bit as well, instead of adding a zero-bit. To do that just shift the bits left and add 1! This method will get us where we want to be.

© 2004 Mr.Mouse and WATTO

The method to calculate the “seed” so far is:

Shift left (size variable) + 1

And our “seed” value will differ from the real “seed” value in the range:

{-2, -1, 0, 1, 2}

Apparently, the encryption process has some way of telling when to add or subtract these values from the result of the shift left + 1 operation, as if looking them up from a table. How would this table look like, and how would it know where to look in the table? The only way to get a hint of this process is by examination of multiple strings and comparing the “seeds” with the size variable, as we will assume that the size variable is also needed to look up the variable from the unknown table (as string 1 and string 2 only have in common the same size, this shows that the encryption uses only this variable to encrypt).

So, make a table from a size variable of 0 upwards. Look at the strings and find the seed, write down the range value it used (-2 or -1 and so forth). Well, perhaps you won’t find strings that are 0 in length, but just fill in those that you do find.

Thus the table will be a single dimensional table like:

Size	Code
0	Code1
1	Code2
2	Code3
3	Code4

…

If you do this, you will discover the following table:

© 2004 Mr.Mouse and WATTO

Size (hexadecimal)	Code
0	-2
1	-1
2	0
3	1
4	2
5	-2
6	-1
7	0
8	1
9	2
A	-2
B	-1
C	0
D	1
E	2
F	-2
10	-1
11	0
12	1
13	2
14	-2
15	-1

Etc.

Let's examine the first string again. It has a size of &h12. That means the code will be 1. The seed will be calculated like this then:

(Shift left (&h12) + 1) + 1 (from file 1) + 1 (Code) = &h27 !

The second string:

(Shift left (&h12) + 1) + 2 (from file 1) + 1 (Code) = &h28 !

The third string:

(Shift left (&h15) + 1) + 3 (from file 1) + -1 (Code) = &h2D !

These are the XOR values that will be used on the first character of the original string. For each subsequent character this value will be incremented by 2. The *complete encryption algorithm* is then:

Painkiller .PAK string encryption:
EncryptedStringCharacter (n) =
OriginalStringCharacter(n) XOR (shift left (string size) + 1 + FileNumber + Code(string size) + n*2)

(*Where n starts at character position 0*)

© 2004 Mr.Mouse and WATTO

Encryption methods are with many, but so are your brain cells

The Painkiller .PAK string encryption is just one of many ways to encrypt and there's no universal tool or process to decrypt all of them. Many archives may encrypt whole resources, while some may use irreversible encryption techniques to address resource names (Microprose .CAT archives from Gunship! and others). As the uncovering of the Painkiller .PAK string encryption should show, there's a lot of guessing and second-guessing needed to solve the puzzle, besides a logic mind. You should train yourself in recognizing logical patterns; think along lines of file structures, bytes and bits. By using a pen and paper you can write down notes, compare things more easily, write down binary values, and try different logical methods to get where you need to be. It really is helpful. In conclusion, the most important ability you need to have to solve encryption techniques is the ability to recognize logical patterns. Good luck!

7. 3 Compression

Like encryption, the subject of *compression* is one we will not be able to address to our full satisfaction. There are many ways to compress (see the appendix for links to websites that cover this subject) and different techniques are needed to effectively compress files of alternating type.
However, we can point out some ways to discover the compression method used.

ZLib compressed files

ZLib (http://www.gzip.org/zlib/) is a free and open source project that is used in many games to compress files and archives, and is comparable in power to others such as RAR or PKZIP. The fact that it's free makes it used a lot, as there are no licensing deals needed. But how do you know if a file is ZLib compressed?

Now, this is probably one of the easiest ones around to spot. Check out the screenshot of HW with an open .TRE file (the format from Star Wars)

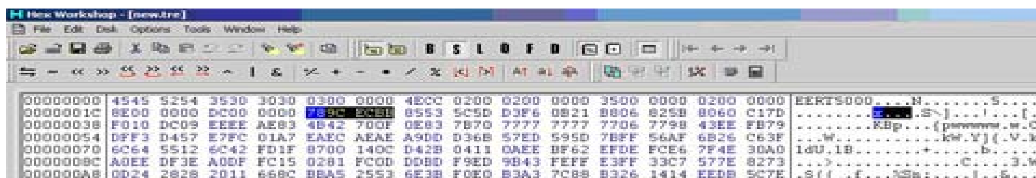

Figure 2. ZLib compression technique can be identified by the 'x' as the first byte.

In the figure, we have highlighted a piece that begins with the letter 'x' (&h78, 120). When you encounter such a piece and you have reason to suspect an individual file begins at that location in an archive of your interest, you may wish to treat it as a ZLib compressed file. You do have to make sure though. Suppose you have figured out that a file starts at a certain offset in an archive,

and in addition you have found variables that tell you the size of this file. Now, just copy and paste the whole putative file into a new file and save it. The next part is tricky as you must have had some experience in unpacking ZLib files. What you need is a simple program that will unpack a standard ZLib-compressed file. In it, you must be able to point to the file and tell it to unpack it. It should then tell you if it was a success or not. Some of these programs will also require you to specify an "uncompressed" size (the original size of the compressed file). With luck, you have already found this variable in the archive as well. Remember to look out for a variable that is slightly greater than the "compressed size" variable when you think you are dealing with compressed files. You can either just create your own program that will try to unpack any given file using zlib.dll, or use existing ones. Check out the http://www.gzip.org/zlib/ website for more information.

PKZip compressed files

Many games nowadays use the 'standard' PKZip compression to compress individual files, and some even just pack all their files into an actual .ZIP file, now and then changing the file extension for their own purposes. Examples of the latter include *Quake 3 .PK3* files, *Thief 2 .CRF* files and *Fall-out Tactics .BOS* files. Some however use the technique itself on individual files and pack them into an archive of their own format. This will not be easy to determine. You may get as far as knowing or at least assuming a resource is compressed, determining with certainty that it is PKZip-compressed is not possible without in depth investigation on the one hand, or a strategy of elimination on the other. This hold true for many if not all compression techniques you wish to identify.

In depth?

One option is to open the executable that is likely to process the GRA in a disassembler. This is extremely time-consuming however if you have not got a vast experience in assembly language. The trick is to trace the code that handles the uncompressing of resources and subsequently *reverse engineer the method!* Naturally, this is not the easiest of tasks. If you wish to go ahead, we recommend a disassembler such as WinDASM (no longer in development, but one of the best around, see if you can find it on the web).

Eliminate them!

Another strategy is to eliminate possible candidate-compression routines by 1. simply trying them on your file and see if they succeed or 2. comparing the structure of you file with those of files that were compressed using a certain technique. Eliminate all those you can find one by one using this strategy until you (hopefully) find the one you are looking for. You should increase your knowledge base on compression techniques as many a game-coder thinks its cool to use some obscure technique from the past.

© 2004 Mr.Mouse and WATTO

Snoop around the executables for clues

You can also use programs such as WinDASM or Hex Workshop to look around the games executables (e.g. .EXE, .COM, .DLL etc.) and see if you identify a piece of text that will tell you more. Sometimes you can find names of functions that are called by the program, or you can find clues in error messages that are saved in the executables. For instance, you could find a 'LZHUncompress' function name, that may point to the .LZH compression technique, or you could find an error message 'RAR: Error in CRC" that tells you .RAR was used. Likewise, some techniques require a licensing deal with the patent holders, so you should examine the credits of a game for a candidate technique (e.g. "Bink Video Compression").

© 2004 Mr.Mouse and WATTO

8. Worked Examples

In this chapter we will go through the process of cracking a format step by step, from opening it for the first time in Hex Workshop to the final format presentation. To do this, we will cover a relatively easy format. In the appendix you will find more complete examples. Let's start off immediately, because the sooner you get to grips with it, the sooner you can start cracking your own!

Quake *.PAK

Games using the Quake (1 or 2) engine save their resources in archive with the .PAK format. How did we tell? Well, when we encounter a new game, and we want to know which files we have to examine we focus on A. large files and B. the title of the file. In the case of Quake we really find only one large file, pak0.pak, and the title and extension clearly indicate it may be some kind of package! We recommend that you download the free Quake2 demo, as the pak0.pak from the demo will be used in this tutorial.

Open the archive and check out the basics:

A. Any GRAIS visible?
B. Determine the size of the archive
C. Locate filename strings if possible
    - If not at the start of the archive, go through the whole archive, starting from the back

When we open the demo's pak0.pak with Hex Workshop we see the following (Figure 2A).

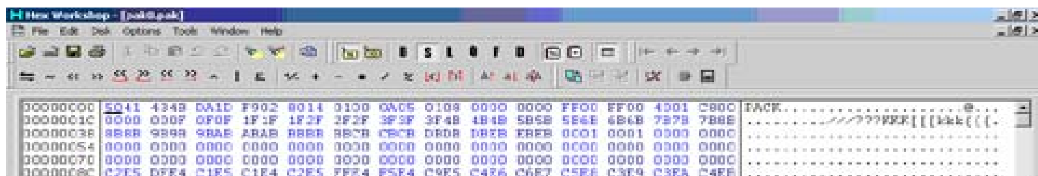

Figure 3A. Start of Quake 2's pak0.pak file in Hex Workshop.

The status bar at the bottom shown the *size of the file* at the bottom right: 49951322 bytes! Also, you can see the cursor is at offset 0. Immediately you notice that the ASCII window shows something interesting: the first 4 bytes make up a plausible English word, "PACK". Could this be our GRAIS? Quite possibly! We select the 4 bytes that make up the word and right-click the selection to get the context menu. Here we select *Add Bookmark…* and write down 'GRAIS' in the description field that we are shown. Before we click *Ok* we set the *Interpret data as…* field to 'string', as we want HW to show the appropriate string value in the Results Window (see User's Bookmarks in Figure 1). Good, next we click *Ok* and you should see something similar to Figure 2B in the results window (by default bottom right).

© 2004 Mr.Mouse and WATTO

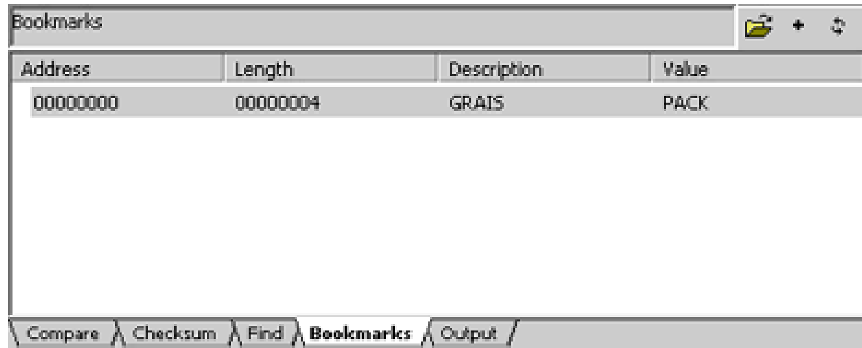

Figure 3B. A bookmark in the Results Window of Hex Workshop.

So now we have determined that the first 4 bytes represent the GRAIS and have book marked it accordingly. We could colour map it as well, to easily find it back on screen. It is up to you if you wish to do that.

We continue to examine the following bytes. We need to find out if there are some filename strings.

We can't see any at the beginning of the file, can we? Apart from PACK we don't see anything that even resembles a string. Then we remember that in many instances resource information, such as filenames, are saved at the back of the archive (in a *Tail*).

Examine the end of the file for filename strings

Simply hit the *End* key on your keyboard to show the end of the pak0.pak file. You should see something like Figure 2C.

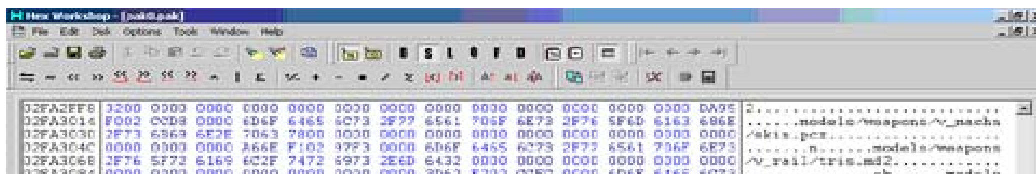

Figure 3C. A snippet from the end of Quake 2's pak0.pak (demo version).

Oh! Check out those strings! Something like *models/weapons/v_machn/skin.pcx* is quite obviously a filename, no doubt about that. And there are many more. But that's not all. What else is there to notice? Well, let's take a closer look at the length of these strings. Just select one string (drag your mouse pointer from the first character of a string to the last while holding the left mouse button).

- TIP: During selection, notice how the status bar at the bottom of HW shows a value for a variable named *Sel:*. This shows the size of the current selection in hexadecimals! As you drag, this value is increased or decreased when you select more or less characters respectively.

© 2004 Mr.Mouse and WATTO

Did you select a string? Read the size of the selection (in effect, the string) in the status bar. Now select another string. Once again, read the size of it. After a couple of times you will notice that these strings have variable sizes. So, the filenames can be any size in this archive apparently. But would this also mean that we have resource entries in this putative tail that are of variable length? Let's check this. As most GRAFs go, filenames and other information such as file offsets and sizes are all saved together into individual entries (blocks). Logic would dictate that each variable would be saved at the same place in the block. So, for argument's sake we'll assume that the program that created the archive saves the filename of a resource first, and then other information about the resource. If this is so, then we can easily see wether these blocks are also of variable length. To do that, first put the cursor on the first character of a filename string. Now select every byte until you reach the first character of the following filename string. In Figure 2D we have done this.

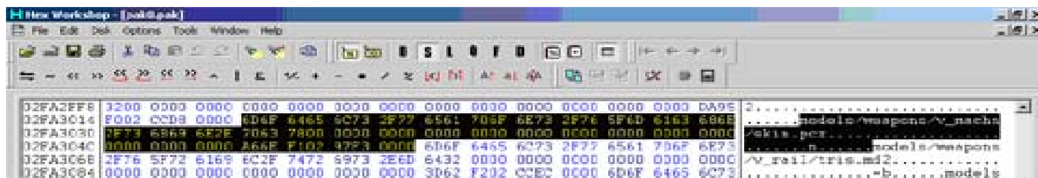

Figure 3D. A block (entry) in the tail of Quake 2's pak0.pak is selected.

We then read the hexadecimal size of this block: 0x40. Or in decimals: 64. When we select multiple entries, each time the size of the entry will be 0x40. This tells us that the size of each block is set, regardless of the filename string, which we have identified as being variable.

Let's go back to the start of the file. Hit *Home* on your keyboard. Before we can go on we need to know how any program that opens .PAK archives knows where to find these filename strings. Well, if it is truly a tail and not just some chunk of the last resource saved in the archive, there are usually two possibilities. First, it may be that the program knows the information is at the back, and perhaps even at a set offset from the end of the file. Second, the program does not know where to find the start of the tail and needs to be told. The variable that points to the address of the tail is for 99% found at the start of the archive. That's why we jumped back to the start! Now let's examine the bytes that come after the PACK string more closely. Set the cursor on the first byte after the 'K'. Look at Figure 2E to compare.

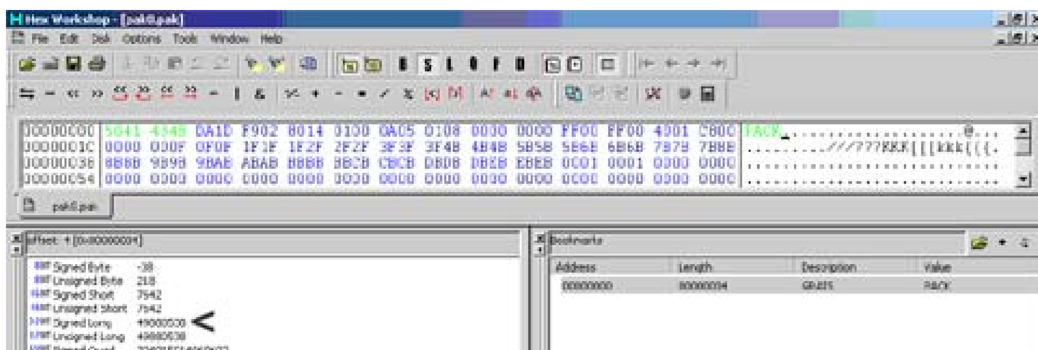

Figure 3E. At the start of pak0.pak. The cursor is at offset 4. The Data Inspector show the relative (interpreted) values at this location.

The bytes are, in sequential order, DA 1D F9 02 80 14 …. Hold on! We remember there are different data types, ranging from 8 bit to 64 bit values. Let's see what we get if we interpret the bytes at this offset as different data types. Look at your Data Inspector. Notice how the values differ per interpretation. Typically, the first four bytes 'look' like something valid. Why? Most variables in an GRA are 32-bit, and the most significant byte (the fourth or right-most byte) in our 32-bit type is a low value. Once you spent a lot of time on this type of puzzling, you will understand why a 4-byte sequence with a low-value fourth byte might raise your attention.
In this case, if we interpret the value at offset 4 as a 32-bit data type, the value is 49880538 (the *Long* datatype).
Now wait a minute, doesn't this value sound familiar? Yes, it does, as the size of the file is 49951322. Our *long* value is less than the size, but only just! Let's see if this might be a pointer to somewhere fun in the GRA. Select the 4 bytes that make up the *long* value. Notice how the Data Inspector only shows 32-bit or less values as you do that. Now, right-click the selected bytes and select *GoTo*->*Offset 0x02F91DDA*. You have jumped to offset 49880538 (see Figure 2F).

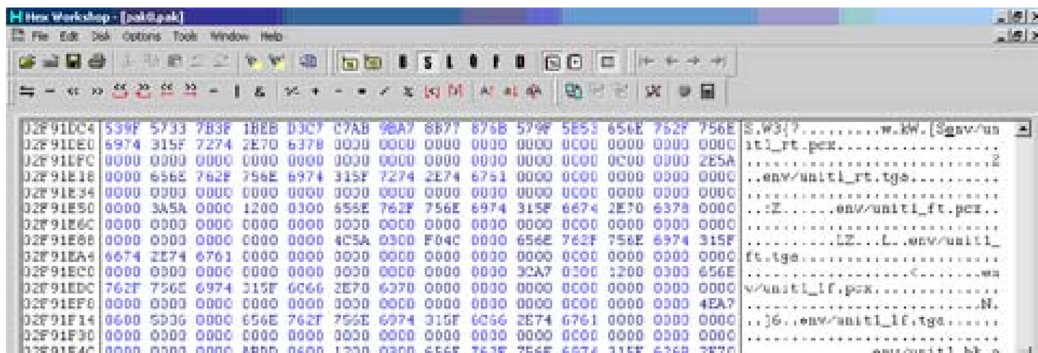

Figure 3F. The cursor is at location 49880538 in pak0.pak.

Bingo! You see the cursor is at the first character of a filename string: *env/unit1_rt.pcx*. Scroll up to see that no other filename strings come before that one. You have found a pointer to a *tail*! Good work!
And as we saw before, this *tail* runs all the way to the end of the file. Let's calculate the *tail size* by subtracting the tail pointer from the size of the archive.

- Tip : Go to the Tools menu and select *Hex Calculator*. This is a handy tool that will let you do basic maths, and convert decimals to hexadecimals and back.

The *tail size* is 70784. Remember that we have previously shown that each entry the tail is 0x40 or 64 bytes in length. Thus we can also calculate the number of files in the file: 70784 / 64 = 1106! This may come in handy if we wish to search for variables such as *number of files* in the GRA.

© 2004 Mr.Mouse and WATTO

For now, let's go back to the *tail pointer*. Hit *Home*. You may bookmark and/or colour map the 4 bytes that represent the *tail pointer* if you wish. Then, set the cursor on the offset to the right of the *tail pointer* (offset 8). Cast your eyes on the Data Inspector. If we interpret the value as a 32-bit *long* we get 70784! We have found the *tail size* variable in the GRA.
The next 32-bit variable at offset 12 is too large to point to anywhere in the file. We can check out some more offsets, to see if there's a 32-bit (or 16-bit) value of 1106 somewhere at the start. After a while, we conclude there's none.

Up to now we have identified a *header* in the pak0.pak (the GRAIS at offset 0, the *tail pointer* at offset 4 and the *tail size* at offset 8). 12 bytes in total. Furthermore, we have found that the *tail* consists of 64-byte entries or blocks with the first variable in these blocks a filename string of variable length.

We still need to do some more investigating of the tail. The tail might reveal other important information about the resources saved in the pak0.pak file. Let's go to the start of the tail again (Figure 2F). See how the first filename entry (*env/unit1_rt.pcx*) is followed by a chunk of 0-value bytes. Apparently, there's some Padding going on. Let's follow the trail of 0's all the way to the second filename string. The trail is broken 56 bytes from the start of the first filename string and 8 bytes before the second filename string. You can see a byte value of 0x0C (12) at this offset, followed by three 0-bytes. 8 bytes could make up two 32-bit values, so let's see what we get if we interpret these 8 bytes like that. The first *long* value is 12, the second *long* value is 23086.
Intuition will tell you that the first may very well be a pointer to the start of the resource (*resource offset*) as it is A. very low and B. we did not find any recognizable variables starting from offset 12 upward. The second long in our tail entry might then represent the *size of the resource*.
We can check all of this easily, especially if we are dealing with an archive that saves entries in the tail in the same order that the actual resources are saved in the archive. First we must make sure we find some *long* values at the end of the following entries in the *tail* as well. This is indeed the case.
Second, we set the cursor at a position 8 bytes before the second filename string (*env/unit1_rt.tga*). The two *long* variables at this offset are 23098 for the putative *resource offset* and 196626 for the *resource size*. Now, subtract the first *resource offset* from the second : 23098 –12 = 23086. See how this is exactly our first *resource size*? We have solved the puzzle! Of course, we will check a bit further up-'stream' to make sure we aren't fooled by coincidence. Having done that we are sure about the following GRAF for pak0.pak:

id Software .PAK (Quake engine)

General outline:

Offset	*Type*	*Description*
0-3	String	GRAIS ('PACK')
4-7	32-bit (*long*)	*Tail pointer*
8-11	32-bit (*long*)	*Tail size*
12-*Tail pointer*	Resource data	Resource Data
Tail pointer-End of file	*N Blocks of 64 bytes*	Resource Information

Tail (single entry):

Relative offset	*Type*	*Description*
0-55	Null-terminated string, 0-padded to size of 56 bytes	*Resource filename*
56-59	32-bit (*long*)	*Resource offset*
60-63	32-bit (*long*)	*Resource size*

If we would want to read from a .PAK file, we could just load tail entries until we reach the end of the file. If we would like to know just how many resources there are in a .PAK file, we simply read the value of the *tail size* and divide it by 64! This may be useful if you wish to reserve memory before you read *tail* entries.

There, we've cracked it!

© 2004 Mr.Mouse and WATTO

9. Appendix

A: Binary → Byte Number Table

Bit 7 (2^7)	Bit 6 (2^6)	Bit 5 (2^5)	Bit 4 (2^4)	Bit 3 (2^3)	Bit 2 (2^2)	Bit 1 (2^1)	Bit 0 (2^0)	Value
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	2
0	0	0	0	0	0	1	1	3
0	0	0	0	0	1	0	0	4
0	0	0	0	0	1	0	1	5
0	0	0	0	0	1	1	0	6
0	0	0	0	0	1	1	1	7
0	0	0	0	1	0	0	0	8
0	0	0	0	1	0	0	1	9
0	0	0	0	1	0	1	0	10
0	0	0	0	1	0	1	1	11
0	0	0	0	1	1	0	0	12
0	0	0	0	1	1	0	1	13
0	0	0	0	1	1	1	0	14
0	0	0	0	1	1	1	1	15
0	0	0	1	0	0	0	0	16
0	0	0	1	0	0	0	1	17
0	0	0	1	0	0	1	0	18
0	0	0	1	0	0	1	1	19
0	0	0	1	0	1	0	0	20
0	0	0	1	0	1	0	1	21
0	0	0	1	0	1	1	0	22
0	0	0	1	0	1	1	1	23
0	0	0	1	1	0	0	0	24
0	0	0	1	1	0	0	1	25
0	0	0	1	1	0	1	0	26
0	0	0	1	1	0	1	1	27
0	0	0	1	1	1	0	0	28
0	0	0	1	1	1	0	1	29
0	0	0	1	1	1	1	0	30
0	0	0	1	1	1	1	1	31
0	0	1	0	0	0	0	0	32
0	0	1	0	0	0	0	1	33
0	0	1	0	0	0	1	0	34
0	0	1	0	0	0	1	1	35
0	0	1	0	0	1	0	0	36
0	0	1	0	0	1	0	1	37
0	0	1	0	0	1	1	0	38
0	0	1	0	0	1	1	1	39
0	0	1	0	1	0	0	0	40
0	0	1	0	1	0	0	1	41
0	0	1	0	1	0	1	0	42
0	0	1	0	1	0	1	1	43
0	0	1	0	1	1	0	0	44
0	0	1	0	1	1	0	1	45
0	0	1	0	1	1	1	0	46
0	0	1	0	1	1	1	1	47
0	0	1	1	0	0	0	0	48
0	0	1	1	0	0	0	1	49
0	0	1	1	0	0	1	0	50
0	0	1	1	0	0	1	1	51
0	0	1	1	0	1	0	0	52
0	0	1	1	0	1	0	1	53
0	0	1	1	0	1	1	0	54
0	0	1	1	0	1	1	1	55
0	0	1	1	1	0	0	0	56
0	0	1	1	1	0	0	1	57
0	0	1	1	1	0	1	0	58
0	0	1	1	1	0	1	1	59
0	0	1	1	1	1	0	0	60
0	0	1	1	1	1	0	1	61
0	0	1	1	1	1	1	0	62

© 2004 Mr.Mouse and WATTO

0	0	1	1	1	1	1	1	63
0	1	0	0	0	0	0	0	64
0	1	0	0	0	0	0	1	65
0	1	0	0	0	0	1	0	66
0	1	0	0	0	0	1	1	67
0	1	0	0	0	1	0	0	68
0	1	0	0	0	1	0	1	69
0	1	0	0	0	1	1	0	70
0	1	0	0	0	1	1	1	71
0	1	0	0	1	0	0	0	72
0	1	0	0	1	0	0	1	73
0	1	0	0	1	0	1	0	74
0	1	0	0	1	0	1	1	75
0	1	0	0	1	1	0	0	76
0	1	0	0	1	1	0	1	77
0	1	0	0	1	1	1	0	78
0	1	0	0	1	1	1	1	79
0	1	0	1	0	0	0	0	80
0	1	0	1	0	0	0	1	81
0	1	0	1	0	0	1	0	82
0	1	0	1	0	0	1	1	83
0	1	0	1	0	1	0	0	84
0	1	0	1	0	1	0	1	85
0	1	0	1	0	1	1	0	86
0	1	0	1	0	1	1	1	87
0	1	0	1	1	0	0	0	88
0	1	0	1	1	0	0	1	89
0	1	0	1	1	0	1	0	90
0	1	0	1	1	0	1	1	91
0	1	0	1	1	1	0	0	92
0	1	0	1	1	1	0	1	93
0	1	0	1	1	1	1	0	94
0	1	0	1	1	1	1	1	95
0	1	1	0	0	0	0	0	96
0	1	1	0	0	0	0	1	97
0	1	1	0	0	0	1	0	98
0	1	1	0	0	0	1	1	99
0	1	1	0	0	1	0	0	100
0	1	1	0	0	1	0	1	101
0	1	1	0	0	1	1	0	102
0	1	1	0	0	1	1	1	103
0	1	1	0	1	0	0	0	104
0	1	1	0	1	0	0	1	105
0	1	1	0	1	0	1	0	106
0	1	1	0	1	0	1	1	107
0	1	1	0	1	1	0	0	108
0	1	1	0	1	1	0	1	109
0	1	1	0	1	1	1	0	110
0	1	1	0	1	1	1	1	111
0	1	1	1	0	0	0	0	112
0	1	1	1	0	0	0	1	113
0	1	1	1	0	0	1	0	114
0	1	1	1	0	0	1	1	115
0	1	1	1	0	1	0	0	116
0	1	1	1	0	1	0	1	117
0	1	1	1	0	1	1	0	118
0	1	1	1	0	1	1	1	119
0	1	1	1	1	0	0	0	120
0	1	1	1	1	0	0	1	121
0	1	1	1	1	0	1	0	122
0	1	1	1	1	0	1	1	123
0	1	1	1	1	1	0	0	124
0	1	1	1	1	1	0	1	125
0	1	1	1	1	1	1	0	126
0	1	1	1	1	1	1	1	127
1	0	0	0	0	0	0	0	128
1	0	0	0	0	0	0	1	129
1	0	0	0	0	0	1	0	130
1	0	0	0	0	0	1	1	131
1	0	0	0	0	1	0	0	132
1	0	0	0	0	1	0	1	133

© 2004 Mr.Mouse and WATTO

1	0	0	0	0	1	1	0	134
1	0	0	0	0	1	1	1	135
1	0	0	0	1	0	0	0	136
1	0	0	0	1	0	0	1	137
1	0	0	0	1	0	1	0	138
1	0	0	0	1	0	1	1	139
1	0	0	0	1	1	0	0	140
1	0	0	0	1	1	0	1	141
1	0	0	0	1	1	1	0	142
1	0	0	0	1	1	1	1	143
1	0	0	1	0	0	0	0	144
1	0	0	1	0	0	0	1	145
1	0	0	1	0	0	1	0	146
1	0	0	1	0	0	1	1	147
1	0	0	1	0	1	0	0	148
1	0	0	1	0	1	0	1	149
1	0	0	1	0	1	1	0	150
1	0	0	1	0	1	1	1	151
1	0	0	1	1	0	0	0	152
1	0	0	1	1	0	0	1	153
1	0	0	1	1	0	1	0	154
1	0	0	1	1	0	1	1	155
1	0	0	1	1	1	0	0	156
1	0	0	1	1	1	0	1	157
1	0	0	1	1	1	1	0	158
1	0	0	1	1	1	1	1	159
1	0	1	0	0	0	0	0	160
1	0	1	0	0	0	0	1	161
1	0	1	0	0	0	1	0	162
1	0	1	0	0	0	1	1	163
1	0	1	0	0	1	0	0	164
1	0	1	0	0	1	0	1	165
1	0	1	0	0	1	1	0	166
1	0	1	0	0	1	1	1	167
1	0	1	0	1	0	0	0	168
1	0	1	0	1	0	0	1	169
1	0	1	0	1	0	1	0	170
1	0	1	0	1	0	1	1	171
1	0	1	0	1	1	0	0	172
1	0	1	0	1	1	0	1	173
1	0	1	0	1	1	1	0	174
1	0	1	0	1	1	1	1	175
1	0	1	1	0	0	0	0	176
1	0	1	1	0	0	0	1	177
1	0	1	1	0	0	1	0	178
1	0	1	1	0	0	1	1	179
1	0	1	1	0	1	0	0	180
1	0	1	1	0	1	0	1	181
1	0	1	1	0	1	1	0	182
1	0	1	1	0	1	1	1	183
1	0	1	1	1	0	0	0	184
1	0	1	1	1	0	0	1	185
1	0	1	1	1	0	1	0	186
1	0	1	1	1	0	1	1	187
1	0	1	1	1	1	0	0	188
1	0	1	1	1	1	0	1	189
1	0	1	1	1	1	1	0	190
1	0	1	1	1	1	1	1	191
1	1	0	0	0	0	0	0	192
1	1	0	0	0	0	0	1	193
1	1	0	0	0	0	1	0	194
1	1	0	0	0	0	1	1	195
1	1	0	0	0	1	0	0	196
1	1	0	0	0	1	0	1	197
1	1	0	0	0	1	1	0	198
1	1	0	0	0	1	1	1	199
1	1	0	0	1	0	0	0	200
1	1	0	0	1	0	0	1	201
1	1	0	0	1	0	1	0	202
1	1	0	0	1	0	1	1	203
1	1	0	0	1	1	0	0	204

© 2004 Mr.Mouse and WATTO

1	1	0	0	1	1	0	1	205
1	1	0	0	1	1	1	0	206
1	1	0	0	1	1	1	1	207
1	1	0	1	0	0	0	0	208
1	1	0	1	0	0	0	1	209
1	1	0	1	0	0	1	0	210
1	1	0	1	0	0	1	1	211
1	1	0	1	0	1	0	0	212
1	1	0	1	0	1	0	1	213
1	1	0	1	0	1	1	0	214
1	1	0	1	0	1	1	1	215
1	1	0	1	1	0	0	0	216
1	1	0	1	1	0	0	1	217
1	1	0	1	1	0	1	0	218
1	1	0	1	1	0	1	1	219
1	1	0	1	1	1	0	0	220
1	1	0	1	1	1	0	1	221
1	1	0	1	1	1	1	0	222
1	1	0	1	1	1	1	1	223
1	1	1	0	0	0	0	0	224
1	1	1	0	0	0	0	1	225
1	1	1	0	0	0	1	0	226
1	1	1	0	0	0	1	1	227
1	1	1	0	0	1	0	0	228
1	1	1	0	0	1	0	1	229
1	1	1	0	0	1	1	0	230
1	1	1	0	0	1	1	1	231
1	1	1	0	1	0	0	0	232
1	1	1	0	1	0	0	1	233
1	1	1	0	1	0	1	0	234
1	1	1	0	1	0	1	1	235
1	1	1	0	1	1	0	0	236
1	1	1	0	1	1	0	1	237
1	1	1	0	1	1	1	0	238
1	1	1	0	1	1	1	1	239
1	1	1	1	0	0	0	0	240
1	1	1	1	0	0	0	1	241
1	1	1	1	0	0	1	0	242
1	1	1	1	0	0	1	1	243
1	1	1	1	0	1	0	0	244
1	1	1	1	0	1	0	1	245
1	1	1	1	0	1	1	0	246
1	1	1	1	0	1	1	1	247
1	1	1	1	1	0	0	0	248
1	1	1	1	1	0	0	1	249
1	1	1	1	1	0	1	0	250
1	1	1	1	1	0	1	1	251
1	1	1	1	1	1	0	0	252
1	1	1	1	1	1	0	1	253
1	1	1	1	1	1	1	0	254
1	1	1	1	1	1	1	1	255

© 2004 Mr.Mouse and WATTO

B. American Standard Code for Information Interchange (ASCII) Table

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	□

© 2004 Mr.Mouse and WATTO

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	Σ
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	φ
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	□

Tables by Alain Courteau, Drummondville, Canada

© 2004 Mr.Mouse and WATTO

C. Format of some common Game Archives

Halo *.MAP

General outline

Position(Bytes)	Type	Description
0 - 3	Long	**Version**: This format only applies to versions 1 and 2.
4 - 7	Long	**TailPointer_Filenames**: The pointer to the *Filenames Tail* (an array of null-terminated strings)
8 - 11	Long	**TailPointer_Fileinformation**: The pointer to the *FileInformation Tail* (each entry for a file is total 12 bytes in length)
12 - 15	Long	**Archive_ItemCount**: The number of items or files stored in the archive
16 - TailPointer_Filenames	Data	*Resources*: The files in the archive
TailPointer_Filenames - TailPointer_Fileinformation	Array of *Filenames Tail*	The names of the files in the archive
TailPointer_Fileinformation - EOF	Array of *FileInformation Tail*	The directory of files

Filenames Tail (Archive_ItemCount)

Position(Bytes)	Type	Description
0 - **NULL**	null-terminated string	**Item_Filename**: The name of the file

FileInformation Tail (Archive_ItemCount)

Position(Bytes)	Type	Description
0 - 3	Long	**Pointer_Filename**: A *relative* pointer to the start of the null-terminated filename for this file. This is relative to the *TailPointer_Filenames* offset
4 - 7	Long	**Item_Size**: The size of the file
8 - 11	Long	**Item_Offset**: The offset to the file

Remarks:
Note that not all *.map files use this format. This is because Halo uses *.map files for 2 purposes: archives, and actual maps. Therefore it is essential that you check the version field equals either 1 or 2.

© 2004 Mr.Mouse and WATTO

Lock On, Midtown Madness 3 *.CDDS

General outline

Position(Bytes)	Type	Description
0 - DDSInfo_Offset	*Header*	**Header**: The header of the archive
DDSInfo_Offset - DDSMIPInfo_Offset	*DDSInfo*	**DDSInfo**: Information on the DDS
DDSMIPInfo_Offset - DDSDataObjects_Offset	*DDSMipInfo*	**DDSMipInfo**: Information on DDS mipmaps
DDSDataObjects_Offset - EOF	*DDSDataObjects*	**DDSDataObjects**: The actual DDS data.

Header

Position(Bytes)	Type	Description
0 - 3	Long	**Unknown**: Has the value 12
4 - 7	Long	**Archive_ItemCount**: The number of mipmaps in the archive
8 - 11	Long	**Unknown**: Has the value 16.
12 - 15	Long	**Unknown**: But is .CDDS file dependent.
16 - 19	Long	**Unknown**: Has the value 0.
20 - 23	Long	**Unknown**: Has the value 0.

DDSInfo (Archive_ItemCount)

Position(Bytes)	Type	Description
0 - 31	String	**Item_Name**: The name of the picture
32 - 35	Long	**Item_Heigth**: The height of the original picture, in pixels
36 - 39	Long	**Item_Width**: The width of the original picture, in pixels
40 - 43	String	**Item_DDSFormat**: An identification string for the type of compression
44 - 47	Long	**Item_HighsizeCount**: I haven't quite figured out the logic of this yet, but this is the number of MIPs in the current collection that are greater than 2048 bytes in size. These also start at a different offset. Generally, the small files are collected at the front of the file, and those larger than 2048 come after the smaller files. Possible something to do with speed of the game, loading of the bigger ones in memory for quick display, while the smaller ones may be read from the file?
48 - 51	Long	**Unknown**: Has the value 0
52 - 55	Long	**Unknown**: Has the value 0
56 - 59	Long	**Unknown**: Has the value 1
60 - 63	Long	**Item_MIPCount**: The number of DDS file data chunks in the current collection
64 - 67	Long	**Item_MIPInfoOffset**: The relative offset of the *DDSMipInfo* entry for the current collection. The offset is relative from this position in the archive.

DDSMipInfo (Archive_ItemCount, Item_MIPCount)

Position(Bytes)	Type	Description
0 - 3	Long	**MIP_RelativeOffset**: The relative offset to the current MIP data chunk
4 - 7	Long	**MIP_Size**: The size of the MIP in bytes (the same as MIP_Width * MIP_Height)
8 - 11	Long	**MIP_Heigth**: The height of the current MIP (in pixels)
12 - 15	Long	**MIP_Width**: The width of the current MIP (in pixels)

© 2004 Mr.Mouse and WATTO

Remarks:

Note again that the MIPs that are in the CDDS are DDS files, *without the 128-bytes DDS-header*. To be able to view them you need to insert the header before the MIP and save it as a new DDS file. Some knowledge of the DDS header structure is needed, but can be looked up on the internet. One way to do it is to rip a header from an existing DDS file (saved in the same surface format, e.g. "DXT5") and insert that before the MIP each time you wish to save. Make sure you change the **height** and **width** variables in the DDS-header (located at byte 13 and 17 respectively) to their corresponding MIP values before you save the file.

Note that the collection of MIPs for each texture start with the MIP in the original dimensions, for example 256 x 64 and decrease in size by half for each subsequent MIP. Thus, you will find that the order of the MIPs in the example will be 256x64, 128x32, 64x16, 32x8, 16x4, 8x2, 4x1, 2x1, 1x1. So the number of MIPs present in each collection is determined by the point where the dimensions of the texture are 1x1. The size of the MIP naturally becomes 4 times smaller each time, *until the size of 16 is reached.* So, although 4x1 = 4, the size will still be 16 bytes.

© 2004 Mr.Mouse and WATTO

Sacrifice *.WAD

General outline

Position(Bytes)	Type	Description
0 - 3	String	**Header**: Identification of this archive format. Has the value "WAD>"
4 - 7	Long	**TailPointer**: Pointer to *Tail*
8 - 11	Long	**TailSize**: Uncompressed size of the tail, in bytes
12 - 19	Unknown	**Unknown**: Unknown meaning
20 - Tail	Data	*Resources:* The data of each file
Tail - EOF	Tail (compressed)	The directory. Note that it is Zlib compressed, so you will need to decompress the directory before reading it.

Tail

Position(Bytes)	Type	Description
0 - 3	String	**Filename**: The name of the file. This string is reversed
4 - 7	String	**FileType**: The type of the file. This string is reversed
8 - 11	Long	**FolderSize**: Total size of all items in this folder
12 - 15	Long	**FolderSize**: Total size of all items in this folder
16 - 19	Long	**NumberOfFiles**: The number of items in this folder
20 - 23	Long	**FolderDivergance**: Has a value of either 0 or 1. The value 1 indicates the start of a new folder. The value 0 indicates a file in the current folder.
24 - 27	Unknown	**Unknown**: Unknown meaning

Remarks:
Note that there is no entry in the tail that represents the offset of each file in the archive. You will have to take the position after the "unknown" long in the header (the first bytes) as the offset of the first file in the list. Subsequent offsets of next files can then be calculated by adding the size of the previous file to the offset of the previous file.

Note that the name of the items is always reversed (for some obscure reason) and that the first 4 bytes in the name are actually a string representing the type of the item (e.g. WAD!, FLDR, TEXT, SAMP), and the second 4 bytes are the name of the item. Note that SAMP files (e.g. in sounds.wad) are actually RIFF (*. WAV) files, preceded with their own 32 bytes header.

© 2004 Mr.Mouse and WATTO

D: Useful References

- *XeNTaX (http://www.xentax.com)*

The website of co-author Mike Zuurman, and home to MultiEx Commander. MultiEx commander is a Windows-based program that can open and manipulate many hundreds of game archives, through use of its own specialist scripting language.

- *WATTO Studios (http://www.watto.org)*

The website of co-author WATTO, and home to Game Extractor. Game Extractor is a game archive viewer/editor that can be run on any platform, and supports a host of different game formats.

- *Wotsit (http://www.wotsit.org)*

A huge collection of file format specifications. The specifications are often taken from the company developing or maintaining the format, so the material is reliable. Contains specifications for all types of files including sounds, images, text, archives, and executables.

© 2004 Mr.Mouse and WATTO

E: Common File Format Tags

Tag	*Extension*	*Type*	*Description*
BM	*.bmp *.dib	Image	Microsoft-standard Bitmap image *(http://www.microsoft.com)*
CWS	*.swf	Animation	Macromedia Flash animation (Compressed) *(http://www.macromedia.com/flash)*
FWS	*.swf	Animation	Macromedia Flash animation (Uncompressed) *(http://www.macromedia.com/flash)*
GIF	*.gif	Image	GIF image *(writing GIF images requires a license from CompuServe)*
MSCF	*.cab	Archive	Microsoft Cabinet archive *(http://www.microsoft.com)*
MZ	*.exe *.dll	Executable	Windows executable program application *(http://www.microsoft.com)*
%PDF	*.pdf	Document	Standard Adobe PDF document / e-book *(http://www.adobe.com)*
PK	*.zip *.gz	Archive	Standard ZIP/GZip archive *(http://www.pkzip.com)*
Rar!	*.rar	Archive	RAR Archive *(http://www.rarsoft.com)*
RIFF	*.wav	Audio	Microsoft-standard audio file *(http://www.microsoft.com)*

© 2004 Mr.Mouse and WATTO

10. Legal Information

This guide aims to help programmers gain an understanding and appreciation of the various file formats in use today. Using this knowledge will help programmers develop their own formats, and increase support for different file formats in their own programs.

This guide supports the exploration of file formats that are open-source or standards. We encourage the exploration of any file, so long as the exploration is for your own benefit only, and will not be used or distributed in an attempt to do anything illegal, including hacking of files, bypassing legal or copyright measures introduced into a file, or for use against a company or individual. If your exploration is for your own benefit and use, we fully support you – there is nothing illegal about exploring the files on your own computer in your own access. If you do wish to use the information you have gained through exploration, make sure you check for any licensing issues, trademarks, or copyrights that may be associated with the format – otherwise you could end up with major fines and criminal charges.

This guide, and the authors, do not encourage or support the exploration of copyright or otherwise protected material for any purpose. The reading of this material does not grant you permission to modify or distribute information contained in any file that is not of your own creation.

This guide, and the authors, do not support and are not affiliated with any game, program, company, copyright, or trademark that is used within. All copyrights, trademarks, and similar rights are used for identification purposes only. All rights reserved.