**All in one Fallout 1-2 code documentation**

Simplified and Edited by Roby Lla

#######################################################

**Background Info:**

This project was made to make modding and further documentation of the Original Fallout Games easier, with all important information centralized in one location.

**Team Members:**

Roby Kalilla, TelamonLivesOn

**Special Thanks:**

Credits to TeamX, Shadowbird, MatuX for various tools and documents

Black Isle Studios for creating the Fallout series of games

**Common Term Definitions:**

Color Palette – A selection of colors indexed in a table that are later referenced

Little Endian – The most significant bit of a byte starts with a 7 and ends with a 0. This is commonly used on PC and Intel-based systems. This is the opposite order of Big Endian

Big Endian – The most significant bit of a byte starts with a 0 and ends with a 7. This is commonly used on Motorola and RISC-based systems. This is the opposite order of Little Endian

BYTE – 8-bit value

WORD – 16-bit value

DWORD – 32-bit value

UInt8 – unsigned 8-bit value

UInt16 – unsigned 16-bit value

UInt32 – unsigned 32-bit value

Int8 – signed 8-bit value

Int16 – signed 16-bit value

Int32 – signed 32-bit value

**File Descriptions:**

.DAT - Archived data lump file for storing game assets.

.FRM - Unpaletted 256-colour image file containing either one or

several frames. Note: the palette for these files is external.

.MSG - **Indexed text** files that contain in-game messages, character

and item dialogues, and are located in the text\english\game and

text\english\dialog folders, respectively.

.ACM – Audio file commonly used by games made/published by

Interplay.

**File Types:**

**-------------\*.FRM---------------**

**Usage -** Used for storing images in both Fallout 1 and 2. Including all animations, except those for splash screens and movies. A FRM file may contain image data for one or all 6 rotations used in Fallout.

**Technical Info –** Uses an index colour model to store the image data, each pixel is represented by an offset into a palette of colours.A colour index of 0 equals to a transparent pixel.

The palette used is often *color.pal*, FRM files can also use their own palette. However, these custom palettes must have the same name as the FRM file. If the file extension is *\*.fr[0-5]* instead of *\**.*FRM* or *\*.frm*, then that file contains image data for the orientation given by the last digit in the extension, otherwise the file contains only orientation 0 or all 6 rotations. This file also contains info to correctly align or center the image data. Each frame has an offset from the previous frame which must be applied before rendering. This offset ensures that in an animation sequence each frame is correctly aligned. The center of the image data is the center of the bottom edge of the frame. To find the position of the top left corner of the frame (needed for rendering):

left = center\_x - (frame\_width / 2)

top = center\_y - frame\_height

Note: Pixel data goes from left to right on the x-axis, and up to down on the y-axis. All data is in the Big Endian format.

|  |  |  |  |
| --- | --- | --- | --- |
| **Header and Data** | | | |
| **Offset** | **Size** | **Possible Value** | **Description** |
| 0x0000 | UInt32 | 0x04 | Version Number of the file |
| 0x0004 | UInt16 | 0x04 | Frames Per Second of the animation |
| 0x0006 | UInt16 | 0x01 -> 0xFF | Frame of the animation on which actions occur (shot, open doors, etc.) |
| 0x0008 | Int16 | 0x01 -> 0xFF | Number of frames per direction |
| 0x000A | Int16 | ? | Required shift in the X direction, of frames with orientation 0 |
| 0x000C | Int16 | ? | Required shift in the X direction, of frames with orientation 1 |
| 0x000E | Int16[4] | ? | Required shift in the X direction, of frames with orientations 2-5 |
| 0x0016 | Int16[6] | ? | Required shift in the Y direction, of frames with orientations 2-5 |
| 0x0022 | UInt32 | ? | Offset of first frame for direction 0 from beginning of frame area. Frame area start from offset 0x003E |
| 0x0026 | UInt32 | ? | Offset of first frame for direction 1 from beginning of frame area |
| 0x002A | UInt32 | ? | Offset of first frame for direction 2 from beginning of frame area |
| 0x002E | UInt32 | ? | Offset of first frame for direction 3 from beginning of frame area |
| 0x0032 | UInt32 | ? | Offset of first frame for direction 4 from beginning of frame area |
| 0x0036 | UInt32 | ? | Offset of first frame for direction 5 from beginning of frame area |
| 0x003A | UInt32 | ? | Size of frame area |
| 0x003E | UInt16 | ? | FRAME-0-WIDTH – the width (in pixels) of frame 0 for orientation 0 |
| 0x0040 | UInt16 | ? | FRAME-0-HEIGHT – the height (in pixels) of frame 0 for orientation 0 |
| 0x0042 | UInt32 | FRAME-0-WIDTH \* FRAME-0-HEIGHT | FRAME-0-SIZE – size (total area) of frame 0. This value is the number of bytes taken up |
| 0x0046 | UInt16 | FRAME-0-WIDTH \* FRAME-0-HEIGHT | Offset in X direction of frame 0 – The offset of this frame from the previous frame |
| 0x0048 | UInt16 | FRAME-0-WIDTH \* FRAME-0-HEIGHT | Offset in Y direction of frame 0 |
| 0x004A | FRAME-0-SIZE (Unsigned) | 0x00 -> 0x3F | Contains the colorIndex for frame 0. Each index is represented by one BYTE, ranging from 0-256. |
| 0x004A + (FRAME-0-SIZE) | VARIABLE (Unsigned) | 0x00 -> 0xFF | Frame data is stored for the rest of the frames at orientation 0. This repeats with orientations up to 5 (if they are present) |

Note: Every item in the game must have an image associated with it.

**-------------\*.DAT---------------**

**Usage –** used in Fallout 1/2 to store game assets such as artwork, critters, scripts, dialogue files, sounds, speech, and more. Fallout uses two \*.DAT files called *master.dat* and *critter.dat*, which are both located in the game’s root folder.

**Technical Info -** There were two different DAT file formats used for the Fallout games. Both Fallout 1 and Fallout 2 used different formats but used the same file ending: \*.dat. To avoid confusion, we'll refer to the Fallout 1 DAT format as DAT1 and the Fallout 2 DAT format as DAT2 in this document. It's important to note that DAT2 is not an extended version of DAT1, but rather a completely rewritten format that shares little in common with DAT1.

Note: *Master.dat* starts with a root node called '.', which contains *color.pal* and several font files. Be careful not to exclude this. All data is in the big endian format.

|  |  |  |  |
| --- | --- | --- | --- |
| **DAT1 Header (General Header)** | | | |
| **Offset** | **Size** | **Possible Value** | **Description** |
| 0x0000 | Int32 | 4 | Directory Count |
| 0x0004 | Int32 | 0x0A (0x5E for master.dat) | Unknown1 (cannot be less than 1, possibly a mem buffer size) |
| 0x0008 | Int32 | 0 | Unknown2 (always 0) |
| 0x000C | Int32 | ? | Unknown3 (Possibly a checksum, appears to work with any value) |
| **DAT1 Header (Directory Name Block, per directory)** | | | |
| 0x0010 | UInt8 | 8 | Number of characters in Directory Name |
| 0x0011 | UInt8[?] | “Data Dir” | Directory Name |
| **DAT1 Header (Directory Content Block, per directory)** | | | |
| ? | Int32 | 4 | Number of files in directory |
| ? | Int32 | 0x0A | Unknown4 (similar to Unknown1, requiring a 1+ value) |
| ? | Int32 | 0x10 | Unknown5 (always 0x10) |
| ? | Int32 | ? | Unknown6 (See Unknown3) |
| **DAT1 Header (File List Block, per file in directory)** | | | |
| ? | UInt8 | 9 | Name length |
| ? | UInt8[?] | “File Name” | Name |
| ? | Int32 | 0x20 | Attributes (0x20 for plain text, 0x40 for LZSS compressed) |
| ? | Int32 | ? | Offset from start of DAT1 file, where file contents begin |
| ? | Int32 | ? | Original, uncompressed file size |
| ? | Int32 | ? | Packed file size (0 is file isn’t compressed) |
| **DAT1 Data (Raw File Data** | | | |
| ? | UInt8[?] | ? | File data for all files (use previous header info to find a specific file/folder) |

Fallout 1 LZSS uncompression algorithm

Originally [written by Shadowbird](https://www.nma-fallout.com/threads/fallout-dat-files.160366/) on NMA forum.

* This is a **decompression algorithm for files compressed with Fallout's LZSS algorithm**, not a file extraction algorithm for getting them out of the DAT file! DAT unpackers already incorporate this.
* It's pretty much a generic LZSS decompression algorithm, with a possible difference from other implementations in that it doesn't prevent overwriting dictionary values while they're being output (see the loop in @FLeven).

DICT\_SIZE = 4096; // Dictionary (a.k.a. sliding window / ring / buffer) size

MIN\_MATCH = 3;

MAX\_MATCH = 18;

Int16 N = 0; // number of bytes to read

Int16 DO = 0; // Dictionary offset - for reading

Int16 DI = DICT\_SIZE - MAX\_MATCH; // Dictionary index - for writing

Byte L = 0; // Match length

Byte FL = 0; // Flags indicating the compression status of up to 8 next characters.

@Start

\* If at the end of file, exit.

\* Read N from input. The absolute value of N is how many bytes of data to read (if N=0, exit).

\* Go to @N<0 or @N>0 accordingly.

@N<0

\* Take the absolute value of N (or multiply N by -1), and write that many bytes directly from input to output (without

putting anything in Dictionary).

\* Go to @Start.

@N>0

\* Clear dictionary (fill with spaces â€” 0x20)

\* DI = DICT\_SIZE - MAX\_MATCH;

\* Go to @Flag.

@Flag

\* Read FL from input.

\* If N bytes have been read from input, go to @Start, otherwise, go to @Next.

@Next

\* If this is the 9th time here since last @Flag, go to @Flag.

\* Go to @FLeven or @FLodd as appropriate.

@FLodd

\* Read 1 byte from input, write it to output and to Dictionary (at position DI).

\* If N bytes have been read from input, go to @Start.

\* DI = DI + 1, or DI = 0 (if past the end of Dictionary).

\* Goto @FlagNext.

@FLeven

\* Read 1 byte from input to DO.

\* If N bytes have been read from input, go to @Start (in a correctly compressed file this should not ever happen).

\* Read L from input.

\* Prepend the high-nibble (first 4 bits) from L to DO (DO = DO | ((L & 0xF0) << 4)) and remove it from L (L = L & 0x0F).

\* (L + MIN\_MATCH) times:

\* Read a byte from dictionary at offset DO (wrap to the start of dictionary if past the end), and write to the output.

\* Write the byte to the Dictionary also, at position DI.

\* DI = DI + 1, or DI = 0 (if past the end of Dictionary).

\* DO = DO + 1.

\* Go to @FlagNext.

@FlagNext

\* Divide FL by 2, rounding down (FL = FL >> 1).

[C# implementation of the above](https://github.com/rotators/Fo1in2/blob/master/Tools/UndatUI/src/dat.cs#L12-L158)

|  |  |  |  |
| --- | --- | --- | --- |
| **DAT2 Data (Data Block of stored files)** | | | |
| **Offset** | **Size** | **Possible Value** | **Description** |
| 0x0000 |  |  |  |
| **DAT2 Header (Number of Files** | | | |
| DataBlock + 0x01 |  |  | Total number of files in the archive |
|  |  |  |  |
|  |  |  |  |

### DAT2 (Little-endian)

DAT2 files are divided in 3 parts, Data Block, Directory Tree and Fixed DAT Information block. Data Blocks contains all the files stored in the DAT, some of them needs to be GZipped, others don't. The Directory Tree contains all the information about each file stored in Data Block, as well as the offset where it's located, if it's compressed or not, packed/unpacked sizes, etc. And finally the Fixed DAT Information block that contains the size in bytes of both full DAT and the Directory Tree. Here you can see a small scheme of how DAT's structure:

| **Part** | **Location** | **Description** |
| --- | --- | --- |
| DataBlock | .............X | Files stored in the archive |
| FilesTotal | X+1 | Number of files in the archive |
| DirTree | X+5.............Z | Number of files in the archive |
| TreeSize | Z+1 | Size of DirTree in bytes |
| DataSize | Z+5 | Full size of the archive in bytes |

* FilesTotal + DirTree corresponds to Directory Tree block
* TreeSize + DataSize corresponds to Fixed DAT Information block

#### The Data Block

The Data Block contains just plain files, their technical information is located in the Directory Tree. Data Block starts from the very beginning of a DAT file. They can be compressed or not, (Fallout engine uses zlib stream data compression), if they're compressed the signature 0x78DA appears at the begin of the file, if not, there is no signature, the file starts without signature. The 0x78DA compression signature has an integer (2 bytes/WORD) nature. 0x78DA in ASCII is "xÃš" as char is 120 for 'x' and 218 for 'Ãš' Compressed files are "zlib stream data" (RFC-1950(zlib format), RFC-1951(deflate format), RFC-1952(gzip format)). However, if you attach this header 1F 8B 08 08 9F E8 B7 36 02 03 to the file, such file could been easily decompressed with any zip file decompression program.

#### The Directory Tree

Directory Tree contains entries that specifies about a file stored in the Data Block. These entries can be varying depending on the FilenameSize of the file (Path + Filename). Like you saw in the scheme located at the beginning of this document, Directory Tree has been divided into 2 parts, FilesTotal and the DirTree. FilesTotal contains how many files are stored in the DAT, DirTree contains all the information about these files. FilesTotal is declared as a DWORD (4 bytes/Long) type and is read in INTEL L-H format. Format of DirTree entries DirTree has a private structure. The length of this structure can vary depending on the length of the Filename (path + filename). All the entries are DWord types unless it's specified. At the end of this chapter you can find a scheme on the structure and the way it's declared on C and Visual Basic programming languages. All the directories and files are stored in DOS 8.3 format, that is 8 characters for the file name and 3 characters for the file extension. All the entries are sorted alphabetically in a descendent direction. Structure scheme: all Dwords are read in INTEL L-H format.

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| FilenameSize | Dword | Length of the ASCII filename. |
| Filename | String | Path and name of the file, For example, "text\english\game\WORLDMP.MSG". The length of the Filename is **FilenameSize**. |
| Type | Byte | 1 = Compressed 0 = Decompressed |
| RealSize | Dword | Size of the file without compression. |
| PackedSize | Dword | Size of the compressed file. |
| Offset | Dword | Address/Location of the file. |

* Dword stands for 4 bytes/long integers 0xNN NN NN NN
* Word stands for 2 bytes integers 0xNN NN
* Byte stands for 1 byte integer 0xNN
* String stands for common string bytes "ABCDEF123456!@#$%/][\", etc.

*Declaration of a DirEntry How to find a DirTreeAddr (starting location of Directory Tree)*

MSG file

- Structure

Each indexed text line has the format: "{}{}{}"

* The first set of curly braces "{}" indicate the line index (number) in decimal format
* The second set of braces (optionally) indicates the name of an ACM sound file
* The third set of braces contain the text to be displayed (dialogue box or observation window)

- Example

Dialogue Line Number

|

|

+----+ ACM sound file name

| |

| |

| +----+ Displayed message

| | (dialogue box or observation window)

| | |

| | |

| | +----+

| | |

v v v

{100} {deadbrahmin001} {You see a horribly mutilated, two-headed cow.

Hamburgers, anyone?}

- Notes

* The length of the lines between the braces cannot exceed 1024 characters
* The length of a single word cannot exceed 53 characters (max width of the in-game dialog window), the rest is not displayed
* Linebreaks inside the braces are allowed however, a trailing portion of the string *must* contain at least one non-space character before the last brace. Otherwise, the game will crash.
* Linebreaks are **ignored** by game engine. That is, if you do
* At the end of the file there is a blank line (with no characters in it)
* The game only looks for text within braces, which allows leaving comments in between the indexed lines
* Comments are usually preceded by a "#", or a pound symbol, unlike Fallout's scripting language, *SSL,* which uses two slashes "//", although some modding tools require comments.
* Sound files are *generally* only used in dialog with 'talking heads', characters with specific voices and special sprites.
* Index is supposed to be unique. If it's not unique, only the latest string with matching index is used by the game.

Note: Fallout Tactics uses .txt files for every text in the game that requires one.

ACM file

This description uses the following integer types:

DWORD 32-bit signed integer

WORD 16-bit signed integer

Also C-like declarations and expressions are used.

ACM file consists of two main parts:

1. short header

2. ACM-Stream, containing packed information

1. Header.

------------

This table describes the meaning of header's fields:

# of bits | Value | Description

-----------+---------------------+-----------------

24 | 0x97 0x28 0x03 | ACM-signature1

8 | 0x01 | ACM-signature2

32 | count of samples | a sample is 16 bit

16 | # of channels | 1-mono, 2-stereo (actually not used)

16 | frequency (bitrate) | 22050 in all known ACMs (actually not used)

4 | packAttrs | (see below)

12 | packAttrs2 |

- Explanation

First 4 bytes of every ACM-file contain 0x01032897, this is kind of

signature and its exact meaning is opaque.

Taking in account that Sig1 and Sig2 are read separately, one can conclude

that Sig1 is an actual ACM-signature, while Sig2 is, for example, a version of ACM.

Number of channels and Bitrate are specified in ACM-file, but they are

not used in the games I know (Fallout1, Fallout2 and Planescape). All the

known at the moment files are 22050 Hz. The # of channels are 1 or 2, but

actual count of channels depends on actual usage of a resulting sound samples.

The next two values specify the behaviour of unpacking algorithm.

The packAttrs denotes the level of recursion in packed-data processing. If this value is zero, no actual data-unpacking is performed.

The packAttrs2 is number of sub-blocks in a Packed-Block (see section 2).

- Actions

Based on header values the next variables are defined (the names correspond to the source code):

a) someSize = 1 << packAttrs (size of one sub-block)

b) someSize2 = someSize \* packAttrs2 (size of one PackedBlock)

c) decBuff\_size = 3\*someSize/2 - 2 (size of "memory"-buffer), this value

is the sum of following geometric progression:

2 + 4 + 8 +...+ ( 1 << (packAttrs-1) )

plus ( 1 << (packAttrs-1) ) once again.

II. The following arrays are created (their names differs from source):

a) PackedBlock = DWORD [someSize2] (corresponds to someBuff). It is more

convenient to think of this array as of variable-length array

DWORD [packAttrs2][someSize] or DWORD [cnt\_of\_blocks][block\_len].

This array is used to process packed data.

b) if packAttrs is not zero, then the following array is created and

initialized with zero: MemoryBuffer = DWORD [decBuff\_size] (corresponds to decompBuff). This array can be thought of as the following structure:

{

WORD [someSize/2][2],

DWORD [someSize/4][2],

DWORD [someSize/16][2],

...

DWORD [1][2]

}

Each item is used at a corresponding recursion level. This buffer is used

as a seam, as a glue between two consecutive PackedBlocks during

data-unpacking.

c) AmplitudeBuffer = WORD [0x10000], and

Buff\_Middle = &amplitudeBuffer[0x8000], the latter is used as an array of

WORDs with indices from -0x8000 to 0x7FFF.

The buffer is filled with amplitude values used in subsequent unpacking.

- ACM-Stream.

This is a bit-stream. It is organized as a collection of bit-blocks,

each of which can be unpacked into someSize2 samples, i.e. into one PackedBlock.

Let's call such a bit-block just a BitBlock. Then an ACM-Stream is just a

sequence of BitBlocks (with various length).

Each BitBlock contains information about amplitudes used in corresponding

PackedBlock and the description of how to manage these amplitudes in order to produce sound samples.

The first 20 bits of BitBlock can be considered as its header:

bits | name

------+------

4 | pwr

16 | val

Using these values the AmplitudeBuffer is filled as following:

count = 1 << pwr

Buff\_Middle [+0] = 0

Buff\_Middle [+1] = val

Buff\_Middle [+2] = 2\*val

...

Buff\_Middle [+(count-1)] = (count-1)\*val

Buff\_Middle [-1] = -val

Buff\_Middle [-2] = -2\*val

...

Buff\_Middle [-count] = -count\*val

PackedBlock is filled with values from AmplitudeBuffer. This is done

with help of special filling subroutines (or Fillers).

There are 14 different Fillers (their description will be given later):

Zero, Ret0, Linear, k13, k12, t15, k24,

k23, t27, k35, k34, k45, k44 and t37.

Each (well, almost each) Filler is intended to fill only one column of

PackedBlock, which is considered as DWORD[packAttrs2][someSize] (n-th column in this case is an array

{ PB[0][n], PB[1][n], ... , PB[pa2-1][n] }).

For each column of PackedBlock one of Fillers is invoked. To select the

Filler, 5 bits are read from BitBlock, and this 5-bit value is used as an index in the following array:

Filler[32] = {Zero, Ret0, Ret0, Linear, Linear, Linear, Linear, Linear,

Linear, Linear, Linear, Linear, Linear, Linear, Linear, Linear,

Linear, k13, k12, t15, k24, k23, t27, k35,

k34, Ret0, k45, k44, Ret0, t37, Ret0, Ret0};

Two parameters are passed to a Filler: its index in the array and the number of a column it is applied to.

Thus the filling of PackedBlock with amplitudes can be outlined with the

following C-pseudocode:

for (int i=0; i<someSize; i++) {

int Ind = get\_bits\_from\_BitBlock (5);

(Fillers [i]) (Ind, i);

}

- Description of Fillers.

Zero. Fills the column with zero. Does not use any bits from BitBlock.

Ret0. Breaks the filling of current PackedBlock, discards its contents and starts a new BitBlock. In rather large ACM-files I've tested I have not found this kind of Filler.

Linear. 'Ind' parameter is the number of bits from BitBlocks which are used as an index of a value in AmplitudeBuffer. In pseudocode:

Linear (int Ind, int column\_n) {

for (int i=0; i<packAttrs2; i++) {

int val = get\_bits\_from\_BitBlock (Ind);

PackedBlock [i][column\_n] = Buff\_Middle [val];

}

}

k13. Uses variable count of bits (up to 3) from BitBlock to fill the column:

bit-sequence | action

(in order of |

appearance) |

--------------+-------------

0 | PB[i][n] = 0; PB[++i][n] = 0

1, 0 | PB[i][n] = 0

1, 1, 0 | PB[i][n] = Buff\_Middle [-1]

1, 1, 1 | PB[i][n] = Buff\_Middle [+1]

k12. Up to 2 bits:

bit-seq. | value

----------+-----------

0 | 0

1, 0 | Buff\_Middle [-1]

1, 1 | Buff\_Middle [+1]

k24. Up to 4 bits:

bit-seq. | value(s)

----------+-----------

0 | 0, 0

1,0 | 0

1,1,0,0 | Buff\_Middle [-2]

1,1,1,0 | Buff\_Middle [-1]

1,1,0,1 | Buff\_Middle [+1]

1,1,1,1 | Buff\_Middle [+2]

k23. Up to 3 bits:

bit-seq. | value

----------+-----------

0 | 0

1,0,0 | Buff\_Middle [-2]

1,1,0 | Buff\_Middle [-1]

1,0,1 | Buff\_Middle [+1]

1,1,1 | Buff\_Middle [+2]

k35. Up to 5 bits:

bit-seq. | value(s) /----> 2bits | B\_M index

--------------+----------- | -------+-----------

0 | 0, 0 | 0,0 | -3

1,0 | 0 | 1,0 | -2

1,1,0,0 | Buff\_Middle [-1] | 0,1 | +2

1,1,0,1 | Buff\_Middle [+1] | 1,1 | +3

1,1,1, 2bits | (\*) -------------/

k34. Up to 4 bits:

bit-seq. | value /----> 2bits | B\_M index

------------+----------- | -------+-----------

0 | 0 | 0,0 | -3

1,0,0 | Buff\_Middle [-1] | 1,0 | -2

1,0,1 | Buff\_Middle [-1] | 0,1 | +2

1,1, 2bits | (\*) -------------/ 1,1 | +3

k45. Up to 5 bits:

bit-seq. | value

-------------+-----------

0 | 0, 0

1,0 | 0

1,1, 3bits | 3bits->B\_M index: 000-> -4, 100-> -3, 010-> -2, 110-> -1

001-> +1, 101-> +2, 011-> +3, 111-> +4

k45. Up to 4 bits:

bit-seq. | value

-----------+-----------

0 | 0

1, 3bits | 3bits->index: 000-> -4, 100-> -3, 010-> -2, 110-> -1

001-> +1, 101-> +2, 011-> +3, 111-> +4

t15. Takes 5 bits from BitBlock. This value is considered as a base-3

number with 3 digits. Each digit is used as an index in Buff\_Middle. So 3

consecutive items of a column are filled.

t27. Reads 7 bits and treats obtained value as base-5 number with 3 digits.

Each digit is used as an index in Buff\_Middle, in this way 3 consecutive

items of a column are filled.

t37. Takes 7 bits and treats their value as base-11 number with 2 digits.

Each digit is used as an index in Buff\_Middle, in this way two consecutive

items of a column are filled.

IV. Processing of data in PackedBlock.

If packAttrs is zero, we do not have do to anything else, we've got the sound samples in PackedBlock. Otherwise we need to apply unpacking algorithm to values in PackedBlock to gain sound samples. This is done in functions unpackValues, sub\_4d3fcc and sub\_4d420c (see source).

All I can tell about the unpacking is that it is recursive. During the

process the PackedBlock is treated as DWORD [cnt\_of\_blocks][block\_len]. At the beginning of the processing

cnt\_of\_blocks = packAttrs2\*2

block\_len = someSize/2.

From some recursion level to the next level cnt\_of\_blocks is multiplied by 2 and block\_len is divided by 2, so the value cnt\_of\_blocks\*block\_len is a constant.

Also I can describe the properties of results from this data processing.

NOTE: to study these properties, an algorithm was slightly altered: in

unpackValues function the for-loop with increment of items of PackedBlock was

commented out. This increment just adds a constant value to all the items of

resulting array, so it is not a core part of algorithm.

Let's call F(PB) the result of applying of functions mentioned above to a

PackedBlock PB. And denote by PB{[i]=v} a PackedBlock where all the items

are equal zero, except for the i-th one, which is assigned a value of v.

That is PB{[1]=5} symbolizes the array {0,5,0,...,0}.

The following properties of F can be observed:

1) "Compact support".

F (PB{[i]=v}) is zero everywhere, except for items from i to

i+2\*someSize-1.

2) Periodicity with period someSize.

F (PB{[i]=v}) and F (PB{i+someSize}=v) differ only in the way, that the

latter one is shifted by someSize.

3) Linearity.

F (PB{[i]=a, [j]=b}) = a\*F (PB{[i]=1}) + b\*F (PB{[j]=1}).

4) Almost orthogonality (a strange one).

Let's denote f\_i = F (PB{[i]=1}), where i=0..someSize-1. w\_i =

F (PB{ [i]=1, [i+someSize]=1, [i+2\*someSize]=1, ... } ), shifted to the left

by (2\*someSize) items (that is we simply discard first items). And group

their indices in the following way:

g0 = {0}

g1 = {1}

g2 = {2, 3}

g3 = {4, 5, 6, 7}

...

g\_packAttrs = { ..., (someSize-1) }

Then any two vectors w\_i and f\_j are orthogonal (in sense of inner product)

if their indices belong to different groups.

Moreover, w\_i and f\_j belonging to the same group are orthogonal if i and j

are of different parity (I am not sure of this term; what I mean to say is

that one of them is even while the another is odd).

May be w\_i can be treated as test-functions for presence of f\_j in

a signal.

For Examples, please open the example directory.