Reverse-engineering MSN Messenger's Video Conversation Formats

Ramiro Polla

1st revision: September 2006

Contents

1 Introduction

This article is an attempt to document the communications protocol used by Video Conversations in MSN Messenger.

The main goal of reverse-engineering this protocol is to have alternative means to decode it, and make it easier for open-source softwares to also implement it and be able to communicate with the official MSN Messenger client.

The data analysed for this article refers to the actual video and audio data that come in TCP or UDP, and does not document all the steps necessary to get there (within a Switchboard Session in MSN Messenger). This kind of information is documented elsewhere on the Internet, and if I have time, I might document them myself.

This article will first document the Transmission of the data. It will provide the details to reconstruct Audio and Video data from the TCP or UDP streams. That reconstructed data can then be fed to a decoder. The second part of this article will deal with the Compression of such data.

2 Transmission

The network layer protocol used in the data I gathered is the Internet Protocol. It uses a high numbered port to send, and low numbered port to receive.

Both parties will necessarily be sending and receiving video, or else MSN Messenger would instead use the Webcam feature, which uses a different kind of communications protocol. Information on this other protocol can be found on the same site you've downloaded this article.

The transport layer protocol used for transferring the Video Conversation data between the sender and the receiver is normally the User Datagram Protocol. Transmission Control Protocol is used mostly if both parties are on the same sub-network.

2.1 UDP

Since UDP does not guarantee a reliable and in-order delivery of packets, MSN Messenger implements a way to re-transmit missing or corrupt information.

Each UDP Frame might contain several Packets. A Packet will not be divided in less than one UDP Frame though. From now on, we will deal with single UDP Packets.

The packet structure is defined by a 10-byte header followed by the payload. The size of the payload is defined by the *size* value in the header. Different kinds of packets (determined by the *code* value) have different ways to deal with the header and data.

The packet structure is defined by the following C type structure:

```
struct vc_udp_header
{
    uint8_t    code;
```

```
    uint16_t   retransmission:5;
    uint16_t   size:11;
    uint8_t    frame_chunk:6;
    uint8_t    nkeyframe:2;
    uint32_t   timestamp;
    uint8_t    frame_number;
    uint8_t    frame_chunks;
};
```

The naming of the variables is based on the way Video Packets use the header. Other kinds of data will have a different use for some of the variables, but the data sizes are the same. The correct meaning of the byte that contains nkeyframe and frame_chunk is not known yet. Having 6 bits for frame chunk seems overkill since no frames will be so huge. Also, the nkeyframe value is probably a 1-bit flag. This structure is *little-endian* and *packed*. Read Annex P.

Some conversations start with 4 or 8 packets that I have not been able to identify the meaning of. Their *code* is either 0x00 or 0x01. I just ignore them for the moment. They don't seem to influence anything in the video or audio data.

Following is an explanation on each variable inside the structure, along with their usual values.

code

This byte will determine what kind of data the payload is. Possible values are:

- *0x44 - 01000100*: Syn/Ack.
- *0x48 - 01001000*: Authorization.
- *0x4a - 01001010*: Audio.
- *0x62 - 01100010*: Video.
- *0x66 - 01100110*: Connection.

Each kind of Packet (determined by the header's code) must be handled differently.

retransmission

This 5-bit value (0-31) is a counter used in re-transmitted packets. If the receiver notices the packet has been corrupted in the transmission, it will request a re-transmission. This value is incremented each time a same packet is sent. If a re-transmitted packet is received, the former packet must be discarded. *Re-transmitted packets always contain the same data, except for this counter.*

size

This 11-bit value (0-2048) determines the size of the payload that follows the header.

frame_chunk

This 6-bit value (0-4096) contains the chunk number in a chunked Video Packet. It start at 0, like in accessing arrays in C.

nkeyframe

This flag (used here as 2-bit, but is probably 1-bit only) is set when a Video Packet is not a keyframe. In other kinds of data, it means nothing, and is set to zero.

timestamp

This double word value is the timestamp (obtained with GetTickCount() by the official MSN Messenger client) in Video Packets. Since Video Packets do not follow a fixed framerate, it is used to keep the correct time.

In Audio Packets, it is a counter to determine which audio frame (starting from 0) it is. Since Audio Packets do have a constant playback time (40ms each), it is not necessary to send an exact timestamp.

frame_number

This byte determines the frame number this current packet belongs to. Using *frame_chunk* and *frame_chunks*, a frame packet can be reconstructed.

It is a counter, and being an unsigned byte, it goes back from 255 to 0. During a bad transfer, you may be receiving data belonging to many frames out-of-order.

frame_chunks

This number, consistent for all the packets with the same *frame_number*, means the number of packets a Video Packet was split up into.

2.1.1 Code 0x44 - Syn/Ack

This Packet is *sent* for every Video Packet *received* as an acknowledgement, and vice-versa.

If you have sent a Video Packet and not received an acknowledgement in a given timeout (not studied by me, but probably just a few seconds), you must send a re-transmission, and expect an acknowledgement for the re-transmission.

The values set in the header are the *size*, which is a multiple of 3, the *frame_chunk*, which is set to 1, and the *frame_chunks*, which is also set to 1. The rest is set to zero.

The payload is made up of 3 bytes for every acknowledged Video Packet. They correspond to the values in the Video Packet to acknowledge, and are made up of this (packed) C structure:

```
struct syn_data
{
    uint8_t    frame_number;
    uint8_t    frame_chunk;
    uint8_t    retransmission;
};
```

```
Example Header: 44 c0 00 01 00 00 00 00 00 01
code           0x44
retransmission 0x00
size           0x0006
frame_chunk    0x01
nkeyframe      0x00
timestamp      0x00000000
frame_number   0x00
frame_chunks   0x01

Example Payload:
0a 03 00 0a 02 01
```

Which is an acknowledgement for the following Video Packets (explained in a while):

```
1st Video Packet Header: 62 80 69 03 9a 66 b3 02 0a 04
2nd Video Packet Header: 62 41 4d 02 9a 66 b3 02 0a 04
```

2.1.2 Code 0x48 - Authorization

This Packet is transmitted a couple of times before the actual Video and Audio data are sent. All its header values are zero, except for *code*. Since *size* is also zero, it contains no payload.

```
Example Header: 48 00 00 00 00 00 00 00 00 00
code           0x48
retransmission 0x00
size           0x0000
frame_chunk    0x00
nkeyframe      0x00
timestamp      0x00000000
frame_number   0x00
frame_chunks   0x00
```

2.1.3 Code 0x4a - Audio

This Packet is transmitted for every 40ms of audio, so you should expect 25 of them per second. Each Audio Packet is 80 bytes long, so the *size* value in the header will be a multiple of 80. It is usually either 80 or 160. When more than one Audio Packet is transmitted at once, the rest are re-transmissions of old packets, in reverse order. The *timestamp* value corresponds to the audio frame number of the new Audio Packet that is being transmitted, and not to its actual time. Remember that each Audio Packet is exactly 40ms long, so it's easy to calculate the exact timestamp. *frame_chunk* and *frame_chunks* are set to 1.

```
Example Header: 4a 00 14 01 f5 02 00 00 00 01
code           0x4a
retransmission 0x00
size           0x00a0
frame_chunk    0x01
nkeyframe      0x00
timestamp      0x000002f5
frame_number   0x00
frame_chunks   0x01
```

In this example, *size* is 160, so there are two Audio Packets. In the payload, the first 80 bytes correspond to a new Audio Packet, which corresponds to *timestamp*. The last 80 bytes are a re-transmission of the Audio Packet that corresponds to (*timestamp* - 1).

In every Audio Packet, there are two Audio Frames. So, in the 80 bytes that make up one Audio Packet, you have 2 Audio Frames, each being 40 bytes. The first Audio Frame is the first 40 bytes, and the second Audio Frame is the last 40 bytes.

2.1.4 Code 0x62 - Video

This Packet is transmitted for every chunk of a Video Frame. Since most Video Frames don't fit in one UDP Packet, they are chunked.

Each packet contains a *frame_number* value to identify which Video Frame they belong to. That value is an unsigned byte, so it will wrap to 0 after it reaches 255. That is not a problem, since you will never need to keep so many frames waiting around to complete transmission. Once you reach, let's say, frame number 50, the previous frame that used the same *frame_number* will be long gone.

In a working Video Conversation, for any given *frame_number*, there should be *frame_chunks* Packets, with the *frame_chunk* value starting at 0 and ending at (*frame_chunks* - 1), just like in an array in C.

In case of a bad connection, retransmission of packets with the exact same *frame_number* and *frame_chunk* will be sent, and they should be used to replace the previous Packet with the same *frame_number* and *frame_chunk*. In case you

receive 2 or more Packets with the same *frame_number* and *frame_chunk*, the ones with smaller *retransmission* value should be discarded.

After you have all the chunks of a frame, you can reconstruct the full Video Frame. Just concatenate the payloads of each Packet belonging to the same *frame_number*, starting from *frame_chunk* 0 up to (*frame_chunks* - 1). An example of such a reconstruction is given in Annex A.

```
Example Header: 62 80 69 00 9c bf 02 00 0a 03
code           0x62
retransmission 0x00
size           0x034c
frame_chunk    0x00
nkeyframe      0x00
timestamp      0x0002bf9c
frame_number   0x0a
frame_chunks   0x03

Example Header: 62 80 69 01 9c bf 02 00 0a 03
code           0x62
retransmission 0x00
size           0x034c
frame_chunk    0x01
nkeyframe      0x00
timestamp      0x0002bf9c
frame_number   0x0a
frame_chunks   0x03

Example Header: 62 c0 63 02 9c bf 02 00 0a 03
code           0x62
retransmission 0x00
size           0x031e
frame_chunk    0x02
nkeyframe      0x00
timestamp      0x0002bf9c
frame_number   0x0a
frame_chunks   0x03
```

In this example, we have the 3 chunks that make up frame number 10. Note that they start with *frame_chunk* 0, and go up to *frame_chunk* 2. The values *nkeyframe*, *timestamp*, *frame_number*, and *frame_chunks* are consistent for all three packets, since they refer to the same Video Frame. The "not keyframe" flag is not set, (!!keyframe == keyframe), so it is a keyframe. The size of the resulting Video Frame is the sum of all Packets' size: 2486.

2.1.5 Code 0x66 - Connection

This Packet is transmitted a couple of times before the actual Video and Audio data are sent. It is the same string used in the Webcam feature, and depends on the information exchanged to create the connection between the sender and the receiver (in MSN Messenger's Switchboard Session).

```
Example Header: 66 60 04 01 00 00 00 00 01 01
Example Payload:
"recipientid=100&sessionid=1347\r\n\r\n"
```

2.2 TCP

TCP is not commonly used. It is mostly if both parties are on the same subnetwork. For this article, I have based myself on one single example file, so it might be wrong for most other files.

The TCP stream contains Audio and Video streams interleaved. They come in small packets with 2-byte headers (*packed*) and corresponding payload.

```
struct vc_tcp_header
{
    uint8_t   size;
    uint8_t   code;
};
```

Packets with code 0x20 are for the Audio Stream, and Packets with code 0x00 are for the Video Stream. I'll deal with them as two separate streams that have no distinction for Frame endings. You should be able to just concatenate all received packets to their specific streams.

Example:

```
Header 1: 56 20
 Payload:  86 bytes
Header 2: c8 00
 Payload: 200 bytes
Header 3: 56 20
 Payload:  86 bytes
Header 4: 2a 00
 Payload:  42 bytes
Header 5: c8 00
 Payload: 200 bytes
Header 6: 56 20
 Payload:  86 bytes
Header 7: b7 00
 Payload: 183 bytes
```

From now on, assume the streams are independent and do not contain *vc_tcp_header*s.

2.2.1 Audio Stream

The Audio Stream is made by 6-byte headers (*packed*) followed by their payloads (usually 80 bytes. I have never seen any other value).

```
struct vc_tcp_audio_header
{
    uint16_t   unk00;
    uint32_t   frame_counter;
}
```

The first value of the header is unknown and constant. The second value is the frame counter of the current frame. The *frame_counter* value is the same as the one named *timestamp* in the UDP Packets.

Just like in the UDP Audio Packets, there are two Audio Frames for each *frame_counter*. So, in those 80 bytes, you have 2 Audio Frames, each being 40 bytes. The first Audio Frame is the first 40 bytes, and the second Audio Frame is the last 40 bytes.

2.2.2 Video Stream

The Video Stream is made by 24-byte headers (*packed*) followed by their payloads. Before each 24-byte header, there is a 1-byte zero value. I don't know what it does, I just ignore it. In this article, I'll just incorporate it in the beggining of the 24-byte header (altough I shouldn't). It makes it simpler to read the stream.

```
struct vc_tcp_video_header
{
    uint8_t    unk00;
    uint16_t   ssize;
    uint16_t   width;
    uint16_t   height;
    uint16_t   nkeyframe;
    uint32_t   size;
    uint32_t   fourcc;
    uint32_t   unk01;
    uint32_t   timestamp;
};
```

This header looks mostly like the header used in the Webcam Feature in MSN Messenger. It contains redundant and somewhat useless information. Its values are the same as the UDP Video Packets header's values, without chunk and retransmission information.

unk00

This is the mentioned unknown zero byte.

ssize

This is the size of the 24-byte struct, which is... 24.

width and height

These are the dimentions of the Video Frame. I am not sure, but I believe they may also be signed. I implemented as unsigned, since I've only seen their values being 320x240.

nkeyframe

This value's first bit is a nkeyframe value, just like in the UDP Video Packets. But there is also more information along in that 16-bit int, which I didn't understand. I just ignore it, and only use the first bit.

size

This is the size of the Video Frame payload that follows right after this structure.

fourcc

This is the FOURCC for the Video. Should always be 'WMV3'.

unk01

This is unknown.

timestamp

This is the Video Frame's timestamp, in milliseconds.

3 Compression

The data that will be dealt with in this section is either an Audio Frame (40 bytes), or a Video Frame (without any header). I'll explain mostly how to get them decoded using Win32 API.

3.1 Audio Frames

Audio Frames are codec'd using "MSN Messenger Audio Codec" aka "Siren7". The file that codec belongs to is *sirenacm.dll*, installed in Windows' system folder by MSN Messenger.

The sample rate of the wave is 16Khz. There are 50 Audio Frames per second.

3.1.1 Analyzing *sirenacm.dll*

Looking at the functions this file provides, you'll see the following:

```
Siren7_DecodeFrame
Siren7_EncodeFrame
Siren7_InitDecoderContext
Siren7_InitEncoderContext
Siren7_SizeofBitstream
Siren7_SizeofDecoderContext
Siren7_SizeofEncoderContext
DriverProc
_Siren7_DecodeFrame@16
_Siren7_EncodeFrame@16
_Siren7_InitDecoderContext@4
_Siren7_InitEncoderContext@4
_Siren7_SizeofBitstream@4
_Siren7_SizeofDecoderContext@0
_Siren7_SizeofEncoderContext@0
```

I'll assume we are not using DriverProc, and interfacing directly with the codec's functions. I have also not spent much time understanding how the EncodeFrame function works. Both formats of each function (stdcall and cdecl) point to the same place. The C prototypes for each function are:

```
int Siren7_DecodeFrame( Siren7_DecoderContext *ctx, void *unk,
                        int32_t *input, float *output );
int Siren7_EncodeFrame( char *p1, char *p2, char *p3,
                        char *p4 ); // Not documented
int Siren7_InitDecoderContext( Siren7_DecoderContext *ctx );
int Siren7_InitEncoderContext( Siren7_EncoderContext *ctx );
int Siren7_SizeofBitstream( int BitstreamType );
int Siren7_SizeofDecoderContext( void );
int Siren7_SizeofEncoderContext( void );
```

Siren7_SizeofDecoderContext and *Siren7_SizeofEncoderContext* just return the value 0x280. This is the amount of memory you need to allocate to create one of such contexts.

Siren7_SizeofBitstream returns 0x28 for *BistreamType* 0x00, 0x3c for *BitstreamType* 0x01, 0x50 for *BitstreamType* 0x02, and 0x00 for anything else. I don't use it, don't know when to use it, and sincerely have no idea what they are, but it was easy to reverse-engineer, so I documented it...

Siren7_InitDecoderContext and *Siren7_InitEncoderContext* will zero out a *Siren7_(De/En)coderContext* and prepare the codec for codec'ing. The memory for the contexts must be allocated by you (and also freed when not needed anymore).

Siren7_DecodeFrame is really annoying. The first argument is a pointer to a *Siren7_DecoderContext* (previously initialized). The second argument is

unknown, but MSN Messenger always passes *NULL*, so that's what I do too. The third argument is the input and it's really tricky, I'll get back to it later. The fourth argument is the output in a 32-bit float waveform. The output size is 0x500 bytes long (and should be allocated by you).

The input is not passed in a friendly manner. The data you receive in one Audio Frame is 40 bytes long, which are really 20 signed little-endian 16-bit ints. The codec expects an input of 80 bytes in 20 signed big-endian 32-bit ints. So you must convert your Audio Frame input. After you convert it, you can pass it to the decoder, and the output will contain 320 audio samples in 32-bit float format.

The following C code/algorithm shows how to initialize a decoder, and get the output from one Audio Frame (half an Audio Packet), in Win32:

```
HMODULE sirenacm;
int (__stdcall *Siren7_DecodeFrame)
    ( Siren7_DecoderContext *, void *, int32_t *, float * );
int (__stdcall *Siren7_InitDecoderContext)
    ( Siren7_DecoderContext *ctx );
int (__stdcall *Siren7_SizeofDecoderContext)
    ( void );

Siren7_DecoderContext *ctx;
int sizeof_ctx;
int16_t *sle16i; // signed little-endian 16-bit ints
int32_t sbe32i[20]; // signed big-endian 32-bit ints
float *output;

// 1. Initialization

sirenacm = LoadLibrary( "sirenacm.dll" );
Siren7_DecodeFrame =
       (int (__stdcall *)
       (Siren7_DecoderContext *, void *, int32_t *, float *))
       GetProcAddress( sirenacm, "Siren7_DecodeFrame" );
Siren7_InitDecoderContext =
       (DWORD (__stdcall *)(Siren7_DecoderContext *))
       GetProcAddress( sirenacm, "Siren7_InitDecoderContext" );
Siren7_SizeofDecoderContext =
       (DWORD (__stdcall *)(void))
       GetProcAddress( sirenacm, "Siren7_SizeofDecoderContext" );

sizeof_ctx = Siren7_SizeofDecoderContext;
ctx = malloc( sizeof_ctx );
Siren7_InitDecoderContext( ctx );

output = malloc( 0x500 );
```

```
// 2. Convert Audio Frame input

// Very Important!
// Signed Little-Endian 16-bit Int: (MSB) 0xabcd (LSB)
// Signed Big-Endian 32-bit Int: (MSB) 0xSXSXcdab (LSB)
// SX is "sign extend"

sle16i = AudioFrame.data;
for( i = 0 ; i < 20 ; i++ )
    sbe32i[i] = bswap_16( sle16i[i] );

// 3. Decode Audio

Siren7_DecodeFrame( ctx, NULL, sbe32i, output );

// output now contains 320 audio samples in 32-bit float

// Repeat steps 2 and 3 for every Audio Frame (in order)

// 4. Deinitialization

FreeLibrary( sirenacm );
free( ctx );
free( output );
```

3.2 Video Frames

Video Frames are codec'd using the "WMV3" codec. The file that codec belongs to is *wmv9vcm.dll*, located in MSN Messenger's folder.

The decoding must start with a keyframe. Subsequent frames can then be fed to the codec for decompressing one by one. To go back one frame, you must find the previous keyframe, and decode from it to the point you want.

In this article, I will give an example on how to decode a Video Frame using VFW API. First, the library must be loaded, then the codec must be opened. After that, frames can be decompressed, and after you're done, you must close the codec, and possibly free the library.

I'm assuming the input and output are 320x240 pixels, but I think using this codec you might be able to define other output sizes. There is an extra 6 bytes of data to be passed to the decoder, which is called the Sequence Layer. I always use these same 6 bytes described below, just like the original MSN Messenger client. I did not check thoroughly what they mean, and I've been told it works with just the first 4 bytes. The output for 320x240 pixels RGB (3 bytes) needs 320 x 240 x 3 = 230400 = 0x38400 bytes of memory.

```
// Include needed headers
```

```
#include <windows.h>
#include <vfw.h>

// Define data that will be needed
struct WMV3BitmapInfoHeaderPlusSequenceLayer
{
    BITMAPINFOHEADER BIH;
    char SequenceLayer[6];
};

char SequenceLayer[] = { 0x0f, 0xf1, 0x80, 0x01, 0x40, 0x0f };
WMV3BitmapInfoHeaderPlusSequenceLayer WMV3;
BITMAPINFOHEADER BITM;
HMODULE wmv9vcm;
FARPROC DriverProc;
HIC hIC;
char *output;

// Initialize data
memset( &WMV3, 0x00, sizeof(WMV3) );
WMV3.BIH.biSize = sizeof(BITMAPINFOHEADER) + 6;
WMV3.BIH.biWidth = 320;
WMV3.BIH.biHeight = 240;
WMV3.BIH.biPlanes = 1;
WMV3.BIH.biBitCount = 0x18;
WMV3.BIH.biCompression = FOURCC_WMV3;
WMV3.BIH.biSizeImage = 0x038400;
memcpy( &WMV3.SequenceLayer, &SequenceLayer,
        sizeof(SequenceLayer) );

memset( &BITM, 0x00, sizeof(BITM) );
BITM.biSize = sizeof(BITMAPINFOHEADER);
BITM.biWidth = 320;
BITM.biHeight = 240;
BITM.biPlanes = 1;
BITM.biBitCount = 0x18;
BITM.biSizeImage = 0x038400;

// Start codec
wmv9vcm = LoadLibrary( "wmv9vcm.dll" );
DriverProc = GetProcAddress( wmv9vcm, "DriverProc" );
ICInstall( ICTYPE_VIDEO, FOURCC_WMV3, (LPARAM) DriverProc,
           NULL, ICINSTALL_FUNCTION );
hIC = ICOpen( ICTYPE_VIDEO, FOURCC_WMV3, ICMODE_DECOMPRESS );
ICDecompressBegin( hIC, &WMV3, &BITM );
```

```
// Assuming "frame" contains a header and the data:
output = malloc( 0x038400 );
WMV3.biSizeImage = frame->header.size;
ICDecompress( hIC, 0, &WMV3, frame->data, &BITM, output );

// "output" now contains the decoded frame
// Do the above steps as many times as you wish...

//Clean up
ICDecompressEnd( hIC );
ICClose( hIC );
ICRemove( ICTYPE_VIDEO, FOURCC_WMV3, 0 );
FreeLibrary( wmv9vcm );
free( output );
```

Annex A: Joining UDP Video Packets

If you're making a list of the whole stream to create an index, it is a good idea to first make a list of all the Packets in the correct order, discarding corrupt packets.

It is best to use a double-linked list, so that you can browse back and forth through the stream. Also, while populating the list, make sure you read through starting by the last entry, and not the first. It is natural that the packets you're receiving belong to the latest frames, so browsing from the end gives better performance.

C++ has the STL to make the job of handling lists much easier, but here I'll show a double-linked list algorithm example to implement the ordering of the Packets received in a double-linked list.

```
// A packet should contain the header and the data.
struct vc_packet
{
    vc_udp_header   header;
    uint8_t   *data;
};

// Start list
first_packet = first_vc_packet( );
last_packet = first_packet;

// Populate it
while( new_packet = next_packet( ) )
{
    for( tmp_packet = last_packet ;
         tmp_packet exists ;
         tmp_packet = tmp_packet->previous )
    {
        // Compare new_packet with tmp_packet
        if both frame_numbers are the same
        {
            if both frame_chunk are the same
            {
                if new_packet's retransmission is greater
                    replace tmp_packet with new_packet
                else
                    discard new_packet
            }
            else if new_packet's frame_chunk is greater
                put new_packet after tmp_packet
        }
        else if new_packet's frame_chunk is greater
```

```
            put new_packet after tmp_packet
    }

    if no replacement ocurred (rare)
    {
        put new_packet before first_packet
        first_packet = new_packet;
    }
}

// All packets are ordered, starting from first_packet
```

After ordering the list, you could go through it removing frames that do not contain all the chunks.

You could then dump the frame data into another struct, which would contain the full frame information in one single place. Chunk and re-transmission information are no longer necessary. Here's an example to extract the data for one given *frame_number*. It uses C's memory function prototypes.

```
struct vc_frame_header
{
    uint8_t    code;
    uint16_t   size;
    uint8_t    nkeyframe;
    uint32_t   timestamp;
    uint8_t    frame_number;
};

struct vc_frame
{
    vc_frame_header header;
    uint8_t *data;
};

// Start frame
vc_frame_header *header = &frame.header;
header->code = first_packet->header.code;
header->size = 0; // Will be incremented for every Packet
header->nkeyframe = first_packet->header.nkeyframe;
header->timestamp = first_packet->header.timestamp;
header->frame_number = first_packet->header.frame_number;
frame.data = NULL;

tmp_packet = first_packet;

while( tmp_packet->header.frame_number == header->frame_number )
```

```
{
    uint16_t newsize = header->size + tmp_packet->header.size;
    frame.data = realloc( frame.data, newsize );
    memcpy( frame.data + header->size, tmp_packet->data,
            tmp_packet->header.size );
    header->size = newsize;
    tmp_packet = tmp_packet->next;
}
```

Annex P: Packed structures

Most structures that deal with data on disk or being transferred on the network have to be carefully implemented. In this document they use, unless otherwise noted, *little-endian* byte ordering. Note: most cross-platform Audio/Video decoding programs don't use structures for this purpose, based on the fact that different systems might have different byte ordering and endianess. Some even have different bit sizes for chars or ints.

Since the structures are normally optimized for space, the data is *1 byte packed*. Pay special attention to the values that do not use up full bytes. To achieve 1 byte packed data using Microsoft's C Compiler, use a *#pragma pack(1)* statement right before the structure declaration. After the declaration, remember to reset the byte alignment, using a *#pragma pack( )* statement.

Example:

```
#pragma pack(1)
struct packed_structure
{
    uint8_t  one_byte;
    uint8_t  three_bits:3;
    uint8_t  five_bits:5;
}; // Totals two bytes
#pragma pack()
```

To achieve the same using GNU's C Compiler, use __attribute__ ((packed)) right before the *;* symbol closing the structure definition.

Example:

```
struct packed_structure
{
    uint8_t  one_byte;
    uint8_t  three_bits:3;
    uint8_t  five_bits:5;
} __attribute__((packed)); // Totals two bytes
```

4 More Info

The studies explained in this article were made so that the program "MSN Webcam Recorder" could support Video Conversations recording and playback. Except for having an exceptionally ugly and messy source code, it works in decoding both Webcam streams and Video Conversation streams. It can be found, along with complete source code, at http://ml20rc.msnfanatic.com