**Background**

   Advanced ( formerly Active ) Streaming Format was developed by [Microsoft](http://www.microsoft.com) in 1995-1998. Its main purpose is to serve as an universal format for storing and streaming media. There are two versions of ASF. Version that is known as 2.0 is well-documented and its specifications are publicly available. Unfortunately, they are not very helpful for developers because this format is not widely used ( if used at all ).  
   On the other hand, there's another version of ASF format ( 1.0 ). It is *extremely* popular. All files with extensions .asf, .asx, .wmv and .wma that you can find in the 'Net are stored in ASF 1.0. Microsoft *never* released any documentation covering this format. There's a rumour that this format is even patented! This situation similar to the one with MPEG-4 specifications: Microsoft appears to take active part in development of specifications for MPEG-4 but does not use these formats in its products, instead, it promotes their closed-source variations ( DivX ;-) and Windows Media Video ).  
   As long as Microsoft does not provide implementations of ASF reader or writer for any platforms except Windows and Macintosh, it is necessary to have at least minimal specification of the format to implement tools for working with ASF 1.0 on all other platforms. This document tries to organize all available information covering the format, received from different sources.  
   Readers are encouraged to get acquainted with ASF 2.0 specifications to better understand the ideas beyond the format and other features that it offers.

**Disclaimer**

This specification was created by analyzing data contained in freely-available media files. No reverse-engineering or other illegal activity took place during collection of this information. Neither author nor any contributors guarantee that any bit of this information is correct.

**Data types**

UINT8, UINT16, UINT32, UINT64 - unsigned integer values, 8, 16, 32 or 64-bit long. In GNU C compiler they are represented by types 'unsigned char', 'unsigned short', 'unsigned long' and 'unsigned long long'.  
FILETIME - unsigned 64-bit integer. Number of 100-nanosecond intervals since midnignt, January 1, 1601, GMT.  
GUID - 128-bit value, that can be generated on any system using special algorithm. The algorithm guarantees uniqueness of any such value ( it means that two different computers or even the same computer in different moments of time cannot generate the same GUIDs ).  
BITMAPINFOHEADER - universal structure that describes format of a ( compressed ) image.

typedef struct

{

long biSize; // sizeof(BITMAPINFOHEADER)

long biWidth;

long biHeight;

short biPlanes; // unused

short biBitCount;

long biCompression; // fourcc of image

long biSizeImage; // size of image. For uncompressed images

// ( biCompression 0 or 3 ) can be zero.

long biXPelsPerMeter; // unused

long biYPelsPerMeter; // unused

long biClrUsed; // valid only for palettized images.

// Number of colors in palette.

long biClrImportant;

} BITMAPINFOHEADER;

WAVEFORMATEX - universal structure that describes format of a ( compressed ) sound stream.

typedef struct

{

short wFormatTag; // value that identifies compression format

short nChannels;

long nSamplesPerSec;

long nAvgBytesPerSec;

short nBlockAlign; // size of a data sample

short wBitsPerSample;

short cbSize; // size of format-specific data

} WAVEFORMATEX;

This structure is immediately followed with an array of bytes of size cbSize.

All time intervals are either measured in 100-nanosecond steps and represented with 64-bit type ( they wrap around each several million years ), or measured in milliseconds and represented with 32-bit ( they wrap around roughly each 49.7 days ) or 16-bit types ( each 65.5 seconds ).

**Basic information**

   ASF 1.0 file consists of 'chunks'. They are similar to chunks from AVI format, but size of their fields was increased.  
Chunk:

|  |  |  |
| --- | --- | --- |
| Field | Type | Size (bytes) |
| Chunk type | GUID | 16 |
| Chunk length | UINT64 | 8 |
| Data | - | Variable |

Chunk type describes type of content in the chunk. See below for list of known chunk type GUIDs.  
Chunk length corresponds to the entire chunk ( i.e. length of data only is chunk length minus 24 ).  
The other important concept is 'packet'. Since the format is supposed to be streamable, all actual data, such as compressed audio or video, is stored in 'packets'. Unlike in ASF 2.0, all packets have fixed size.  
Each valid file should contain at least two chunks. They are File Header Chunk and Data Chunk. File Header Chunk contains all the information required to start processing actual data, while Data Chunk contains data packets.

**Headers**

File Header chunk:

|  |  |  |
| --- | --- | --- |
| Field | Type | Size (bytes) |
| Chunk type | GUID | 16 |
| Chunk length | UINT64 | 8 |
| Number of subchunks | UINT32 | 4 |
| Unknown | - | 2 |
| Chunks | - | Variable |

This chunk is special because it contains other chunks in the data field. There may be any number of such chunks, but we need to know about two special kinds of them.  
  
Header Object:

|  |  |  |
| --- | --- | --- |
| Field | Type | Size (bytes) |
| Chunk type | GUID | 16 |
| Chunk length | UINT64 | 8 |
| Client GUID | GUID | 16 |
| File size | UINT64 | 8 |
| File creation time | FILETIME | 8 |
| Number of packets | UINT64 | 8 |
| Timestamp of the end position | UINT64 | 8 |
| Duration of the playback | UINT64 | 8 |
| Timestamp of the start position | UINT32 | 4 |
| Unknown, maybe reserved ( usually contains 0 ) | UINT32 | 4 |
| Flags ( usually contains 2 ) | UINT32 | 4 |
| Minimum size of packet, in bytes | UINT32 | 4 |
| Maximum size of packet | UINT32 | 4 |
| Size of uncompressed video frame | UINT32 | 4 |

Value 0x02 in flags probably means that the file is seekable.  
Minimum & maximum sizes of packet are typically equal. It is not precisely known how to handle ASF file if it's not true.  
Stream Object:

|  |  |  |
| --- | --- | --- |
| Field | Type | Size (bytes) |
| Chunk type | GUID | 16 |
| Chunk length | UINT64 | 8 |
| Stream type (audio/video) | GUID | 16 |
| Audio error concealment type | GUID | 16 |
| Unknown, maybe reserved ( usually contains 0 ) | UINT64 | 8 |
| Total size of type-specific data | UINT32 | 4 |
| Size of stream-specific data | UINT32 | 4 |
| Stream number | UINT16 | 2 |
| Unknown | UINT32 | 4 |
| Type-specific | - | Variable |
| Stream-specific | - | Variable |

Type-specific data is data which meaning can be derived only from stream type. It may be followed by fields that also depend on value of audio error concealment type.  
Second unknown value in this object seems to be absolutely random, but if there is more than one stream in the file, they all hold the same value here.  
Type-specific data for video stream:

|  |  |  |
| --- | --- | --- |
| Field | Type | Size (bytes) |
| Picture width | UINT32 | 4 |
| Picture height | UINT32 | 4 |
| Unknown | UINT8 | 1 |
| BITMAPINFOHEADER size | UINT32 | 4 |
| Picture format | BITMAPINFOHEADER | Variable |

Field 'Picture format' usually contains BITMAPINFOHEADER structure, which is 40 bytes long, but it is not a good idea to rely on this fact, since it may contain something of a larger size.  
  
Type-specific data for audio stream:

|  |  |  |
| --- | --- | --- |
| Field | Type | Size (bytes) |
| Sound format | WAVEFORMATEX | 14 |
| Sound format extension | - | Variable |

Size of sound format extension is equal to cbSize member of WAVEFORMATEX structure.  
  
Stream-specific data for audio stream:

|  |  |  |
| --- | --- | --- |
| Field | Type | Size (bytes) |
| H, Total number of audio blocks in each scramble group | UINT8 | 1 |
| W, Byte size of each scrambling chunk | UINT16 | 2 |
| Block\_align\_1, usually = nBlockAlign | UINT16 | 2 |
| Block\_align\_2, usually = nBlockAlign | UINT16 | 2 |
| Unknown | UINT8 | 1 |

This data is only present if 'Audio error concealment type' field in the main structure contains corresponding GUID. See section 'Audio error concealment' for details on this field.  
  
All valid ASF files contain one Header Object, as well as one Stream Object per stream.

**Data chunk**

Data chunk:

|  |  |  |
| --- | --- | --- |
| Field | Type | Size (bytes) |
| Chunk type | GUID | 16 |
| Chunk length | UINT64 | 8 |
| Unknown | GUID | 16 |
| Number of packets | UINT64 | 8 |
| Unknown | UINT8 | 1 |
| Unknown | UINT8 | 1 |
| Packets | - | variable |

As mentioned above, packets have fixed size. It can be found in the corresponding field of Header Object.

**Packets**

   Compressed video and audio data are usually organized into 'frames' or 'objects' of an arbitrary size. When one needs to transfer such data in packets of a fixed size, there can be three opportunities:  
a) Frame size is close to the size of the packet. It would be acceptable to store the frame completely in one packet and pad it to needed size.  
b) Frame is larger than the packet. Then it needs to be 'fragmented' into several fragments and sent in different packets.  
c) Frame is significantly less than the packet. In this case it would be a good idea to send multiple frames in the same packet. It is called 'grouping'.  
<Packet>: <Header> <Segment> [<Segment>] ... <Padding>  
There may be several formats of headers, but packets in most movies start with the V82\_Header:

|  |  |  |
| --- | --- | --- |
| Field | Type | Size (bytes) |
| 0x82 | UINT8 | 1 |
| Always 0x0 (?) | UINT16 | 2 |
| Flags | UINT8 | 1 |
| Flags are bitwise OR of: 0x40 Explicit packet size specified 0x10 16-bit padding size specified 0x08 8-bit padding size specified 0x01 More than one segment | | |
| Segment type ID | UINT8 | 1 |
| Packet size | UINT16 | 0 or 2 ( present if bit 0x40 is set in flags ) |
| Padding size | Variable | 0, 1 or 2 ( depends on flags ) |
| Send time, milliseconds | UINT32 | 4 |
| Duration, milliseconds | UINT16 | 2 |
| Number of segments & segment properties | UINT8 | 0 or 1 ( depends on flags ) |

Precise meaning of 'packet size' is not known. It rarely appears in ASF streams, and when it does, it shows complete length of data in this packet ( from the beginning of packet header to the end of the last segment ). Sometimes it's OR'ed with 0x10 or 0x8, but I've never seen packets with specified nonzero padding size and 0x40 set in flags.  
Segment:

|  |  |  |
| --- | --- | --- |
| Field | Type | Size (bytes) |
| Stream ID | UINT8 | 1 |
| Sequence number | UINT8 | 1 |
| Segment-specific fields | - | Variable |

Most significant bit ( 0x80 ) is set in the stream ID if the segment contains a keyframe.  
Here things become a bit more complicated. Segment-specific fields depend on whether this segment is grouped ( i.e. it contains more than one frame ) or not. This can be deduced from flags value, which is inside segment-specific fields itself!  
  
Segment-specific fields, no grouping:

|  |  |  |
| --- | --- | --- |
| Field | Type | Size (bytes) |
| Fragment offset | UINT8, UINT16 or UINT32 | Variable |
| Flags | UINT8 | 1 |
| Object length | UINT32 | 4 |
| Object start time, milliseconds | UINT32 | 4 |
| Data length | UINT8 or UINT16 | 0, 1 or 2 |
| Data | - | Variable |

"Fragment offset" is offset of this fragment in the object ( e.g. video frame ) that contains it. For complete frame in the fragment, fragment offset is 0 and data length is equal to object length.  
"Flags" can be either 0x01 or 0x08. 0x01 means "grouping ( multiple objects in segment )", and 0x08 means "no grouping ( single object or fragment )".  
"Data length" field is not needed if this segment is the only one in the packet, because in this case data takes all remaining space in the packet ( of course, taking padding into account ). Thus, it's only present when bit 0x01 is set in packet flags.  
"Fragment offset" field size is determined by 'Segment Type ID' packet header value. Known possible values for the latter are 0x55, 0x59 and 0x5D, which correspond to 1, 2 and 4 byte sizes.  
"Data length" field size is determined by 'Number of segments' packet header value. When 'Number of segments' field is present, its lower bits ( probably 6 of them ) contain number of segments, set bit 0x40 means that 'Data length' segment field is 1-byte wide, and set bit 0x80 means that 'Data length' segment field is 2-byte wide. Otherwise, this field size defaults to 2 bytes.  
  
Segment-specific fields, grouping:

|  |  |  |
| --- | --- | --- |
| Field | Type | Size (bytes) |
| Object start time, milliseconds | UINT8, UINT16 or UINT32 | Variable |
| Flags | UINT8 | 1 |
| Unknown | UINT8 | 1 |
| Data length | UINT16 | 0 or 2 |
| Repeat until we run out of data length: | | |
| Object length | UINT8 | 1 |
| Data | - | Variable |
| ... | | |

This structure is similar to the one with 'no grouping', but it does not have 'fragment offset' field, because fragmentation and grouping can not take place simultaneously.  
Each segment has a field called 'sequence number'. It can be used to reassemble fragmented objects. Subsequent objects have sequence numbers that differ by 1 ( there will be larger skips in 'sequence number' fields when grouping takes place ). Different fragments of the same object have the same sequence number and the same object start time.  
Packets are usually organized in order of increasing timestamps. It is not known if it's always true. Packets may be missing, and this case should be properly handled.

**Audio error concealment**

   Sometimes compressed audio is stored in stream in a special 'scrambled' manner. It should be descrambled before passing data do audio decompressor. This technique is supposed to increase stream tolerance to errors.  
All audio data is separated into 'audio blocks'. Size of an audio block is a multiple of data sample size. The process is defined with two variables: audio block length ( Width ) and number of audio blocks in 'scrambling chunk' ( Height ). This process is most simple to demonstrate with the picture.

Data sent to decoder: [0] [4] [8] [1] [5] [9] [2] [6] [10] [3] [7] [11]

width=4

height=3

[0] [1] [2] [3]

[4] [5] [6] [7]

[8] [9] [10][11]

Data stored in the stream: [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]

Here each [x] is data region with size specified in Block\_align\_1 field of scramble definition structure. Width is first field of that structure, and Height is second field, divided by third.  
When total amount of data is not multiple of 'scrambling chunk' size ( in bytes, that's first field times second field ), the remaining part is written as is, without scrambling.  
Even when GUID in the stream header indicates that audio is scrambled, there may be no need in it, because very often values of W or H are equal to 1.

**Streaming over the Internet**

   Media content in ASF format can be streamed over the Internet in several ways. Most popular way is streaming using HTTP protocol. Other protocols, such as UDP, may be supported as well.  
URLs for ASF files may lead to 'redirectors'. Redirector is a XML file that describes media that it refers to, includes other URLs and additional data needed for stream playback. Redirector files often have extensions .asx, but it's probably not a requirement. Some details can be found at <http://msdn.microsoft.com/peerjournal/wm/g060199a.asp>.

**Streaming using HTTP protocol**

ASF URLs that start with http:// or mms:// refer to streams that are delivered to end-user over protocol that's based on HTTP. They can consist of redirectors, pre-recorded or live ( broadcast ) data. To start transmission, client program connects to server using TCP ( often on port 80 ), sends a HTTP request and listens for data.  
Here are descriptions of HTTP requests, in sprintf()-compatible form.  
  
The initial HTTP request of media player. It is used to query for the media type header of the stream (needed for checking if the codecs are installed at the client and for obtaining the type of stream (live stream, pre-recorded content etc..) . Note that the request-context changes with every new HTTP request:  
  
"GET %s HTTP/1.0\r\n", filename   
"Accept: \*/\*\r\n"  
"User-Agent: NSPlayer/4.1.0.3856\r\n"  
"Host: %s\r\n", server\_name  
"Pragma: no-cache,rate=1.000000,stream-time=0,stream-offset=0:0,request-context=1,max-duration=0\r\n"  
"Pragma: xClientGUID={c77e7400-738a-11d2-9add-0020af0a3278}\r\n"  
"Connection: Close\r\n\r\n"  
  
The HTTP request that starts downloading prerecorded (=seekable) content. The stream-offset parameter defines the start offset in the ASF file on the server. The stream-time is the timecode (milliseconds) for seeking within the stream:  
  
"GET %s HTTP/1.0\r\n", file  
"Accept: \*/\*\r\n"  
"User-Agent: NSPlayer/4.1.0.3856\r\n"  
"Host: %s\r\n", server\_name  
"Pragma: no-cache,rate=1.000000,stream-time=0,stream-offset=%u:%u,request-context=2,max-duration=%u\r\n", offset\_hi, offset\_lo, length  
"Pragma: xPlayStrm=1\r\n"  
"Pragma: xClientGUID={c77e7400-738a-11d2-9add-0020af0a3278}\r\n"  
"Pragma: stream-switch-count=%d\r\n", num\_streams  
"Pragma: stream-switch-entry=%s\r\n", stream\_selection  
"Connection: Close\r\n\r\n"  
  
Pay some attention to lines with 'stream-switch-count' and 'stream-switch-entry'. First line includes a number of streams which you want to receive. Second line includes a string in the following form:  
ffff:1:0 ffff:2:2 ffff:4:2 ( etc. )  
where each entry corresponds to one stream, first value is always 'ffff', second value is the stream ID from ASF header and third value is unknown.  
Even if you request for only selected streams, server may send you all of them. So, request with num\_streams=1 and stream\_selection="ffff:1:0" will sometimes give you all streams ( instead of one ). Same rules apply to broadcast request, described further.  
This is the HTTP request that starts downloading live (=broadcast) content.  
  
"GET %s HTTP/1.0\r\n", file  
"Accept: \*/\*\r\n"  
"User-Agent: NSPlayer/4.1.0.3856\r\n"  
"Host: %s\r\n", server\_name  
"Pragma: no-cache,rate=1.000000,request-context=2\r\n"  
"Pragma: xPlayStrm=1\r\n"  
"Pragma: xClientGUID={c77e7400-738a-11d2-9add-0020af0a3278}\r\n"  
"Pragma: stream-switch-count=1\r\n"  
"Pragma: stream-switch-entry=ffff:1:0\r\n"  
"Connection: Close\r\n\r\n"   
  
Server reply on these requests consists of an arbitrary number of lines which are terminated by \n ( 0x0A ) or \r\n ( 0x0D 0x0A ) ( HTTP header ), an empty line and actual content.   
First line of HTTP header has form:  
"HTTP/1.%d %d %s", version, errorcode, string  
where version is 0 or 1, errorcode is 3-digit HTTP error code and string is an optional server message. Possible error codes include 200 - no error, 404 - file not found, and others.  
Other important HTTP header lines:  
"Content-Type: %s", content\_type  
  Content type of data. Possible values:  
  application/octet-stream - 'real' binary ASF stream.  
  audio/x-ms-wax, audio/x-ms-wma, video/x-ms-asf, video/x-ms-afs, video/x-ms-wvx, video/x-ms-wmv, video/x-ms-wma - ASX redirectors.  
"Pragma: features=%s",features  
  If "features" has substring "broadcast", the stream is live ( not prerecorded ).  
Headers are followed by actual content, separated into chunks. However, these chunks are different from the ones described in previous sections.

|  |  |  |
| --- | --- | --- |
| Field | Type | Size (bytes) |
| Basic chunk type | UINT16 | 2 |
| Chunk length | UINT16 | 2 |
| Sequence number | UINT32 | 4 |
| Unknown | - | 2 |
| Chunk length confirmation | UINT16 | 2 |
| Body data | - | Variable |

Chunk length corresponds to data that starts from sequence number field.  
Basic chunk type can be 0x4424 ( Data follows ), 0x4524 ( Transfer complete ) and 0x4824 ( ASF header chunk follows ).  
For type 0x4824 'body data' should be parsed according to the same rules as a local ASF file. It is arranged so that ASF recorder program would not need to leave any 'holes' in file while recording - this chunk includes all ASF content up to the beginning of first packet with compressed media.  
For type 0x4424 'body data' contains a complete packet ( for example, first byte of this data is usually 0x82 ). Network transmission may send chunks that are shorter than pktsize from ASF file header, by chopping off padding section.  
Some fields in ASF file header may be empty, especially for the live stream.

**Known GUIDs**

struct GUID

{

long v1;

short v2;

short v3;

unsigned char v4[8];

int operator==(const GUID& guid) const{return !memcmp(this, &guid, sizeof(GUID));}

};

/\* GUID indicating audio stream header \*/

const GUID guid\_audio\_stream=

{ 0xF8699E40, 0x5B4D, 0x11CF, 0xA8, 0xFD, 0x00, 0x80, 0x5F, 0x5C, 0x44, 0x2B };

/\* GUID indicating video stream header \*/

const GUID guid\_video\_stream=

{ 0xBC19EFC0, 0x5B4D, 0x11CF, 0xA8, 0xFD, 0x00, 0x80, 0x5F, 0x5C, 0x44, 0x2B };

/\* GUID indicating that audio error concealment is absent \*/

const GUID guid\_audio\_conceal\_none=

{ 0x49f1a440, 0x4ece, 0x11d0, 0xa3, 0xac, 0x00, 0xa0, 0xc9, 0x03, 0x48, 0xf6 };

/\* GUID indicating that interleaved audio error concealment is present \*/

const GUID guid\_audio\_conceal\_interleave=

{ 0xbfc3cd50, 0x618f, 0x11cf, 0x8b, 0xb2, 0x00, 0xaa, 0x00, 0xb4, 0xe2, 0x20 };

/\* GUID for header chunk \*/

const GUID guid\_header=

{0x75B22630, 0x668E, 0x11CF, 0xA6, 0xD9, 0x00, 0xAA, 0x00, 0x62, 0xCE, 0x6C};

/\* GUID for data chunk \*/

const GUID guid\_data\_chunk=

{0x75b22636, 0x668e, 0x11cf, 0xa6, 0xd9, 0x00, 0xaa, 0x00, 0x62, 0xce, 0x6c};

/\* GUID for index chunk \*/

const GUID guid\_index\_chunk=

{0x33000890, 0xe5b1, 0x11cf, 0x89, 0xf4, 0x00, 0xa0, 0xc9, 0x03, 0x49, 0xcb};

/\* GUID for stream header chunk \*/

const GUID guid\_stream\_header=

{0xB7DC0791, 0xA9B7, 0x11CF, 0x8E, 0xE6, 0x00, 0xC0, 0x0C, 0x20, 0x53, 0x65};

/\* ASF 2.0 header \*/

const GUID guid\_header\_2\_0=

{0xD6E229D1, 0x35da, 0x11d1, 0x90, 0x34, 0x00, 0xa0, 0xc9, 0x03, 0x49, 0xbe};

/\* File header object \*/

const GUID guid\_file\_header=

{0x8CABDCA1, 0xA947, 0x11CF, 0x8E, 0xE4, 0x00, 0xC0, 0x0C, 0x20, 0x53, 0x65};

**Credits**

Most of the information contained in this document was collected by Avery Lee <uleea05 at umail.ucsb.edu> and by unknown author of ASFRecorder program. Translated from C/C++ into readable English by yours, truly <divx at euro.ru>. Comments and improvements are welcome.

Last modified on April 5, 2001