OpenDML AVI File Format Extensions

Version 1.02

OpenDML AVI M-JPEG File Format Subcommittee

Last revision: February 28, 1996

Reformatting: September 1997

Table of Contents

Table of Contents

1.0 Introduction

The OpenDML File Format Subcommittee is defining an ***AVI-compatible*** file format that addresses the particular needs of professional video. As such, this document relates specific proposals to these needs.

Further work by OpenDML will elaborate on extensions to the Codec and Vidcap specifications of Video for Windows to meet the needs of professional video.

Scope

This document describes the proposed format of the OpenDML-compliant AVI extensions, specifically for the Motion-JPEG DIB AVI file. This format is an extension to the AVI M-JPEG DIB as defined in the *JPEG DIB FORMAT* technical note from Microsoft. The format is based on the ISO 10918 defined JPEG format.

It is assumed that the reader is familiar with JPEG as defined in the ISO 10918 document. For additional information on JPEG, see the *ISO 10918 Information technology* -- *Digital compression and coding of continuous-tone still images: Requirements and guidelines.*

For additional information about the JPEG DIB format, see Microsoft Multimedia Technical Note *JPEG DIB FORMAT*.

For additional information about RIFF files see the Microsoft Windows Software Development Kit *Multimedia Programmer’s Guide* and *Multimedia Programmer’s Reference*.

For additional information on installable compressors and decompressors, see the *Video Compression/Decompression Drivers* technical note from Microsoft.

Portions of the above Microsoft documents have been reprinted with permission of Microsoft Corp.

OpenDML Sub-Committee on File Format: High Level Goals

- Interoperability and AVI Compatibility
- Large File Support
- Frame and Field Indexing
- Interleaving Audio with Video
- Source and Header Information Storage
- OpenDML File Format Certification Code
- Microsoft Certification and Communication

Version History

Version 1.0	November 13, 1995	First Release
Version 1.1	December 18, 1995	Includes small changes from Microsoft

1.0 Overview of Profession Video Requirements

Goal

Provide interoperability between different hardware and software vendors' Motion JPEG codecs. Support Video for Windows AVI file format for MJPG video, while improving the performance of these systems.

Issues

AVI: Support Current AVI Format

The goal is to support interoperability by standardizing on the AVI file format for Motion-JPEG video. This implies a minimum compatibility to support the standard AVI file access interface, as well as the vidcap and codecs that are part of Video for Windows. This means support of the MJPG DIB specification, supporting the RIFF file format as well as the standard AVI stream headers, allowing for optional audio interleaving, and supporting one storage of one frame of video per ##dc data chunk in the LIST 'movi' chunk of the AVI file.

60 Fields Per Second vs. 30 Frames Per Second; or 24 Frames Per Second For Film

Professional Video Applications require the ability to sequence individual fields of the video data for certain playback rates and video effects. For example, in order to do a slow motion effect, it is not sufficient to just repeat the frames; the fields must be individually repeated. For example, as sequence of fields in a file numbered 123456... should be played at half speed as 11223344... rather than 12123434.... As such, the individual fields should be accessible. AVI requires that frames are stored per data chunk. Storing individual fields in each chunk would break AVI applications.

The extended AVI format will allow for access to individual fields. In addition, the format will allow for storage of 24 frame based files, such that file readers can properly convert between 24 frame-based data storage and 25 or 30 frame PAL and NTSC playback.

Improved Tombstone and Header Information

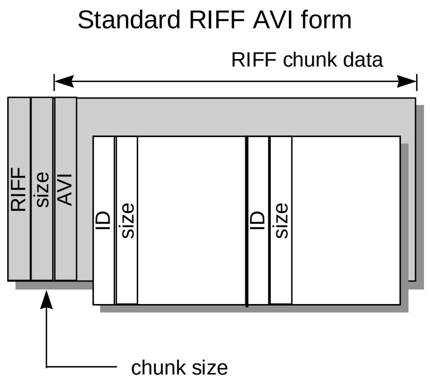
Professional video applications need extended information such as starting timecode and reel ID to be contained inside the AVI file. More advanced information such as the location of timecode discontinuities, etc. are also desirable.

2.0 Increased AVI File Size

Introduction: RIFF Chunk Format

The AVI file begins as a standard RIFF chunk (note that this is a non-standard representation that shows the 4-byte size):

```
RIFF (size) 'AVI ' ...
```



There is only one RIFF chunk per file. RIFF sub-chunks may be either LIST chunks or regular sub-chunks. The LIST chunk obeys the same structure and may have regular or LIST sub-chunks; all other RIFF sub-chunks have just an ID identifier and a size. These regular sub-chunks may not have sub-chunks.

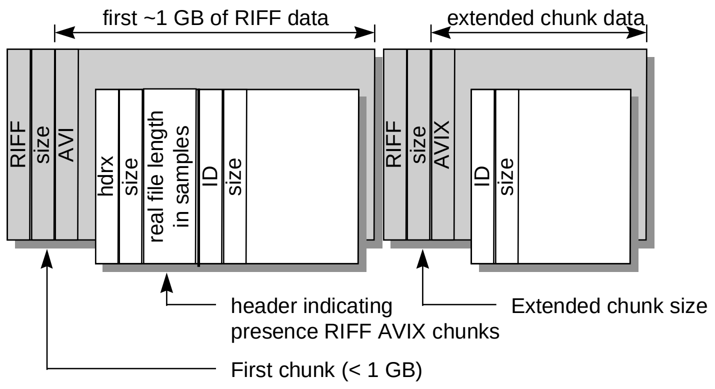
Increased File Size Limits (>> 1 GB)

The current RIFF file format implies a maximum chunk size of 4 GB because the size is stored as a 32-bit value. However, limitations to the RIFF parser code and MCIAVI limits the file to only 1 GB. At data rates of 10 MB/s, a 1 GB file will last less than 2 minutes.

Extension for File Size > 1 GB

An AVI file can be extended beyond 1 GB by placing more than one RIFF chunk in the same file (which is legal). Each RIFF chunk can have a maximum size of 1 GB. Standard AVI applications will see the first RIFF chunk (RIFF 'AVI") as a standard AVI file, and this chunk should be complete. Information that the file is extended over 1 GB should be placed in the Extended AVI header (see below). Subsequent RIFF chunks will be identified by the AVIX (for AVI extended) chunk id. These RIFF 'AVIX' chunks do not need to contain further data. It is expected that the AVIX chunks only contain LIST 'movi' data. All other AVI information should be stored in the first (RIFF 'AVI') chunk so that it is accessible by all applications.

RIFF AVI form for files > 1 GB

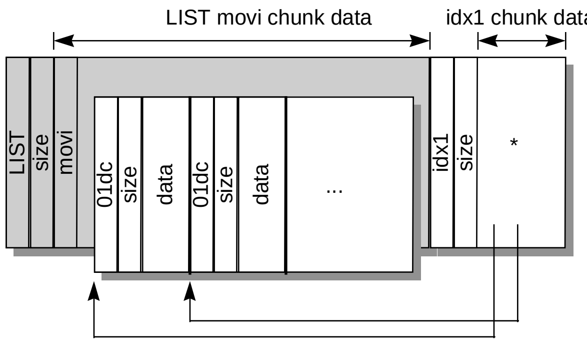


3.0 Frame and Field Indexing

Introduction: 'idx1' Index Chunk

The AVI file format specifies an optional 'idx1' chunk, which contains a list of the offsets and chunk types of every chunk inside the LIST 'movi' chunk. This list is used to make AVI playback and seeks more efficient because the location of each frame of video can be found without scanning through each sub-chunk of the LIST 'movi' data.

Standard idx1 form



* data of idx1 chunk is an index to each movi sub-chunk, includes offset and length of each chunk

Issues

Allow Field Indexing

In order to support field accurate indexing while still retaining AVI compatibility, the video data must be stored one frame per ##dc 'movi' chunk, while an index must be present that gives the locations of both fields (if they exist) within the frame. Applications could then use field-based codecs to perform field-based effects (e.g., slow motion, etc.).

Allow Incremental Growth of Files

The 'idx1' chunk follows the LIST 'movi' chunk in the file. As such, it must be moved in order to insert new data into the file. An interleaved frame index would be more efficient for the purposes of file growth.

Minimize Disk Seeks

In playback, there are several causes of inefficiency. One is that the index is at the end of the file, requiring a large seek when opening the file. The other is that the index itself contains all the chunks inside the 'movi' data, not just those for one AVI stream. As such, the index must be preprocessed so that the location of each frame in the index can be found.

Proposed Index List

Overview

Instead of using an index chunk like ‘idx1’, a new index structure is proposed. The index has the following main characteristics:

- The can either be a single index chunk, or a two tiered index, with a super index pointing to interleaved index segments in the ‘movi’ data.
- The index can include locations of the fields within a frame.
- The index has no restrictions to 4 GB file size.
- There is an index per data stream (if necessary).

The advantages of the index are as follows:

- The index can contain the locations of the individual fields in each chunk as well as the chunk information as a whole. The ‘idx1’ chunk only contains indices of the frames.
- The index segments are interleaved within the ‘movi’ data, meaning that the file can be grown. The ‘idx1’ chunk is present at the end of the file. It must be moved for the file to grow.
- The index segment is smaller than an ‘idx1’ chunk, so the amount of data needed to be read in order to access a particular frame is less. Also, if the index is interleaved before its data (i.e., less efficient in write), then playback will be more efficient than the ‘idx1’ chunk.
- The index lists are present per stream number (##). As such, the list can be accesses like an array, unlike the ‘idx1’ chunk, which contains entries for all ‘movi’ chunks from all streams.

The index is implemented as an index of indexes (optional) and an index of chunks. Both are based on a base index form.

Base Index Form ‘indx’

Thus the actual implementation is based on a base index form ‘indx’:

```
struct _aviindex_chunk {
   FOURCC fcc;
   DWORD  cb;
   WORD   wLongsPerEntry;   // size of each entry in aIndex array
   BYTE   bIndexSubType;    // future use.  must be 0
   BYTE   bIndexType;       // one of AVI_INDEX_* codes
   DWORD  nEntriesInUse;    // index of first unused member in aIndex array
   DWORD  dwChunkId;        // fcc of what is indexed
   DWORD  dwReserved[3];    // meaning differs for each index
                            // type/subtype.   0 if unused
   struct _aviindex_entry {
      DWORD adw[wLongsPerEntry];
   } aIndex[ ];
};
```

The actual size of the entries of the `aIndex` array is entered into `wLongsPerEntry`. Between the `cb` field and `wLongsPerEntry` the actual size of the array is known. The field `nEntriesInUse` allows a chunk to be allocated longer than the actual number of used elements in the array.

The types defined for the `bIndexType` and `bIndexSubtype` field are the following:

```
// bIndexType codes
//
#define AVI_INDEX_OF_INDEXES 0x00   // when each entry in aIndex
                                           // array points to an index chunk
#define AVI_INDEX_OF_CHUNKS  0x01   // when each entry in aIndex
                                           // array   points to a chunk in the
file
#define AVI_INDEX_IS_DATA       0x80       // when each entry is aIndex is
                                           // really the data
// bIndexSubtype codes for INDEX_OF_CHUNKS
//
#define AVI_INDEX_2FIELD     0x01   // when fields within frames
                                           // are also indexed
```

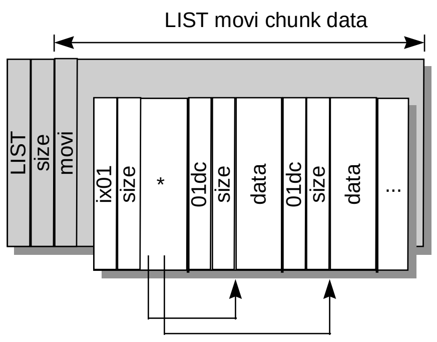
AVI Standard Index Chunk

The AVI Standard Index chunk contains information that indexes AVI frames.

```
typedef struct _avistdindex_chunk {
   FOURCC   fcc;             // ’ix##’
   DWORD    cb;
   WORD     wLongsPerEntry;  // must be sizeof(aIndex[0])/sizeof(DWORD)
   BYTE     bIndexSubType;   // must be 0
   BYTE     bIndexType;      // must be AVI_INDEX_OF_CHUNKS
   DWORD    nEntriesInUse;   //
   DWORD    dwChunkId;       // ’##dc’ or ’##db’ or ’##wb’ etc..
   QUADWORD qwBaseOffset;    // all dwOffsets in aIndex array are
                             // relative to this
   DWORD    dwReserved3;     // must be 0
   struct _avistdindex_entry {
      DWORD    dwOffset;     // qwBaseOffset + this is absolute file offset
      DWORD    dwSize;       // bit 31 is set if this is NOT a keyframe
   } aIndex[ ];
} AVISTDINDEX, * PAVISTDINDEX;
```

A single standard index chunk can only index data within a 4 GB region. The `dwOffset` field points to the start of the data itself, and not to the start of the RIFF chunk for that field.

Standard Index Chunk



* data of Standard AVI index chunk is an index to
each movi chunk data, includes offset and length of chunk.

The index chunk above is shown in the 'movi' data, but may be found in the stream header (see below).

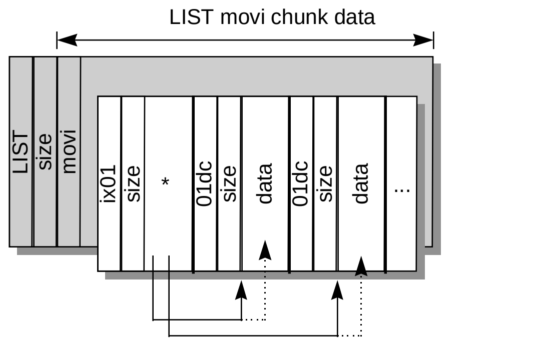
AVI Field Index Chunk

The AVI Field Index Chunk is the same as the Standard Index Chunk except that it contains the locations of each field in the frame.

```
typedef struct _avifieldindex_chunk {
   FOURCC   fcc;               // 'ix##'
   DWORD    cb;
   WORD     wLongsPerEntry;    // must be 3 (size of each entry in
                               // aIndex array)
   BYTE     bIndexSubType;     // AVI_INDEX_2FIELD
   BYTE     bIndexType;        // AVI_INDEX_OF_CHUNKS
   DWORD    nEntriesInUse;     //
   DWORD    dwChunkId;         // '##dc' or '##db'
   QUADWORD qwBaseOffset;      // offsets in aIndex array are relative to this
   DWORD    dwReserved3;       // must be 0
   struct _avifieldindex_entry {
      DWORD    dwOffset;
      DWORD    dwSize;         // size of all fields
                                 // (bit 31 set for NON-keyframes)
      DWORD    dwOffsetField2; // offset to second field
   } aIndex[  ];
} AVIFIELDINDEX, * PAVIFIELDINDEX;
```

The field `wLongsPerEntry` is set to 3 because of the addition of the `dwOffsetField2` field in the array. This is indicated by setting the sub type to `AVI_INDEX_2FIELD`.

Field Index Chunk



* data of Field Index Chunk is an index to each movi sub-chunk data, includes offset and length of chunk, as well as offset to the second field.

The index chunk above is shown in the movi data, but may be found in the stream header (see below).

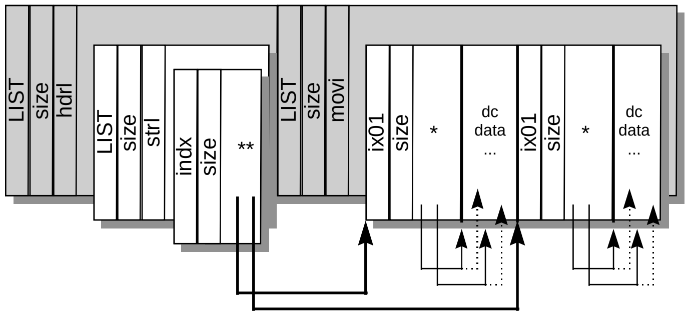
AVI Super Index Chunk

The Super Index Chunk is an index of indexes and is always found in the ‘indx’ chunk of an AVI file. It is defined as follows:

```
typedef struct _avisuperindex_chunk {
   FOURCC  fcc;            // ’ix##’
   DWORD   cb;             // size of this structure
   WORD    wLongsPerEntry; // must be 4 (size of each entry in aIndex array)
   BYTE    bIndexSubType;  // must be 0 or AVI_INDEX_2FIELD
   BYTE    bIndexType;     // must be AVI_INDEX_OF_INDEXES
   DWORD   nEntriesInUse;  // number of entries in aIndex array that
                           // are used
   DWORD   dwChunkId;      // ’##dc’ or ’##db’ or ’##wb’, etc
   DWORD   dwReserved[3];  // must be 0
   struct _avisuperindex_entry {
      QUADWORD qwOffset;   // absolute file offset, offset 0 is
                           // unused entry??
      DWORD    dwSize;     // size of index chunk at this offset
      DWORD    dwDuration; // time span in stream ticks
   } aIndex[ ];
} AVISUPERINDEX, * PAVISUPERINDEX;
```

The `bIndexSubType` is set to the type of index that the Super Index points to. If the index chunks are Standard Index Chunks, then the value should be 0. If the index chunks are AVI Field Index Chunks, then the value should be `AVI_INDEX_2FIELD`. This implies that a stream cannot mix Field and Standard Index Chunks.

Super Index Chunk



* data of ##ix chunk is an index to each movi sub-chunk data, includes offset and length of chunk, and possibly the offset to the second field.
** data of Super Index indx chunk is an index to each ##ix movi sub-chunk.

If we use the ##pc chunks, then they must be indexed by their own Index List: Palette changes should not appear in the same Index List as ##dc chunks, since they eliminate the ability to access an element (frame) directly without having to scan the list.

Index Locations in RIFF File

Unlike the ‘idx1’ chunk, a single index is stored per *stream* in the AVI file. An ‘indx’ chunk follows the ‘strf’ chunk in the LIST ‘strl’ chunk of an AVI header. This ‘indx’ chunk may either be an index of indexes (super index), or may be an index to the chunks directly. In the case of video, this means that the chunk is either a `AVISUPERINDEX` or an `AVIFIELDINDEX`/`AVISTDINDEX`.

If the ‘indx’ chunk is a standard or field index chunk (i.e., not an index of indexes) then the stream has only one index chunk and there is none in the ‘movi’ data.

If the ‘indx’ chunk is a Super Index, then the corresponding index chunks are marked with ‘ix##’ in the ‘movi’ data. The ## is the stream number, the same as for the ##dc or ##wb. The index chunks can be either standard or field index chunks.

A file can be easily grown if it has a standard index in the ‘indx’ chunk position. The chunk can be moved to a new ‘ix##’ chunk, and a new super index can be inserted into the stream header (‘indx’ position). New ‘ix##’ chunks can be added to grow the file.

Note the reversal of the ‘##ix’ to ‘ix##’. This is for AVI backward compatibility. Note that INDEX_IS_DATA streams remain as ‘##ix’.

Here is a sample RIFF file in the format shown by the RIFFWALK utility:

```
RIFF (7F038718) 'AVI '
     LIST (0001084C) 'hdrl'
         avih (00000038)
```

```
        LIST (000044E0) ’strl’
            strh (00000038)
            strf (00000428)
            indx (00003FF8)    <- video superindex
            vprp (00000064)    <- video property header
        LIST (0000405E) ’strl’
            strh (00000038)
            strf (00000012)
            indx (00003FF8)    <- audio superindex
        LIST (0000404C) ’strl’
            strh (00000038)    <- timecode stream...
            strf (00000000)
            indx (00003FF8)    <- timecode superindex
        LIST (00000060) ’odml’
            dmlh (00000054)    <- size still temp.
    LIST (7F000000) ’movi’
        01wb (00002dF0)
        00db (00030004)
        01wb (00002dF0)
        00db (00030004)
        ... time passes
        02ix (00003FF8)     <- data chunk for timecode stream
        ... time passes
        ix00 (00003FF8)     <- partial video index
        ix01 (00003FF8)     <- partial audio index
        .. etc ..
    idx1 (00030000)         <- original index (optional)
RIFF (7F038718) ’AVIX’
```

4.0 Interleaving Audio with Video

Issues

Interleaving Rate

Files for Editing

Files for Playback

Single or Multiple Files for Video and Audio

Multiple Tracks of Audio Storage

Mono and Stereo Audio

Support of multiple streams of data exists in AVI. A proposal to recommend that audio be stored as two mono streams should be examined. Not all applications would be able to play this back in stereo, however.

Audio / Video Time Stamping (Sync Point List)

It is necessary to add time stamping information to account (and correct) for drift between audio and video when using hardware that is not able to maintain a perfect synchronization between audio and video (via a hardware connection) and when there exists a drift in the crystal frequency on audio hardware.

This can be accomplished by adding a sync point list into the file. A sync point list identifies two samples in different streams that are to be played at the same time. Thus, drift can be corrected by identifying sync points and if necessary, correcting for the drift during playback.

Notes on Interleaving Audio and Video

With the exception of AVI files playing off CD-ROM drives, it is not necessary to interleave the audio and video into REC chunks. In fact, with higher video and audio data rates, it is probably more efficient to store the audio in large consecutive blocks. As such, the audio can be interleaved in large blocks (not necessarily done on a frame basis).

Audio data chunks may or may not be in integral frames, or multiple of frames. The other option would be a constant number of bytes. In addition, audio may be interleaved before or after the corresponding video. This is just a tradeoff between record and playback performance. Typically, saving audio chunks after video in record will give better record performance, which is important because hard drives tend to be less performant during write operations. However, during playback, multiple streams and cuts may be played, reducing the drive performance.

5.0 Source and Header Information Storage

Goal

The goal is to implement a standard way to store certain header and tombstone information in an AVI file so that other applications can interpret and use this data. In many cases, the header data is not uniformly set between different AVI files.

Required Information

Main AVI Header (avih)

Total Frames

The `dwTotalFrames` field indicates the size (number of frames) within the first RIFF 'AVI' chunk.

AVI Stream Header (strh)

Quality Information

AVI defines the quality of a clip by a DWORD value from 0 to 10000. This value seems arbitrary. For MJPG files, a standard quality should be defined. The quality can be defined by either of these three modes:

- constant average data rate
- constant Q-table
- lossless

Lossless is not part of the Baseline process. As such it requires the definition of a new Process. Under baseline, lossless can be achieved by setting the Q-tables to unity[1].

Under constant data rate, the desired data rate for the file should be specified (in kb/s). This, plus the number of fields per frame, should be able to determine the Q-tables required to compress the image.

---

[1] While a lossless (or pseudo lossless) process can be defined by setting the Q-tables to unity, the ISO specification of lossless JPEG compression is a different process. Lossless encoding as defined by ISO does not use DCT transformed data. For a sample to be encoded, a predictor is formed from the reconstructed values of up to three neighborhood samples. This gives a prediction of the of the sample to be encoded. The prediction is subtracted from the actual value of the sample and the difference is losslessly entropy coded using either Huffman or arithmetic coding. As far as forming a type of pseudo lossless compression from a baseline JPEG process with unity Q tables, there is the potential of some slight loss of data during the DCT process due to arithmetic precision. Practically speaking, there is no real image quality loss, but the word *lossless* implies a guarantee of reconstructing the source image on a bit by bit basis.

Under constant Q-table, there are two possibilities. Either the absolute Q-tables are given (8x8 matrices) per component (in our case, 2), or a factor which is used to generate any Q-table of a standard one. The constant Q-table file would of course have the same tables repeated per field, as per the MJPG DIB spec. The advantage of a Q-factor is the reduction of header information. However, tweaked Q-tables would no longer be possible.

Scale / Rate

The scale and rate parameters will define the correct ratio for different video standards. Known ratio values are (rate/scale):

NTSC	30000/1001
PAL	25/1

rcFrame

The on-disk RECT coordinates are 16-bit values. The RECT structure in NT is a 32-bit (LONG) for each coordinate, but the on-disk ones remain 16-bit.

AVI Stream Format (strf)

Height

The MJPG DIB spec lists that heights less than 288 are for single frames (one chunk) and greater than 288 are for interleaved fields. However, there is no real connection between frame height and interleaving. The **biHeight** parameter refers to the raw height of the frame (interleaved or not). OpenDML codecs that operate in field by field modes will be passed this value divided by 2 to get the field height.

Extended AVI Header (dmlh)

```
typedef struct {
   DWORD dwTotalFrames;
} ODMLExtendedAVIHeader;
```

Total Frames

The `dwTotalFrames` field indicates the **real** size of the AVI file. Since the same field in the Main AVI Header ‘avih’ indicates the size within the first RIFF ‘AVI’ chunk.

Video Properties Header (vprp)

The video properties header identifies video signal properties associated with a digital video stream in an AVI file. This header attempts to address two main video properties:

- The type of video signal (PAL, NTSC, etc., as well as the resolution of the video signal).
- The framing of the compression within a video signal.

The parameters can be used to uniquely describe a video signal.

```
typedef struct {
   DWORD              VideoFormatToken;
   DWORD              VideoStandard;
   DWORD              dwVerticalRefreshRate;
   DWORD              dwHTotalInT;
   DWORD              dwVTotalInLines;
   DWORD               dwFrameAspectRatio;
   DWORD               dwFrameWidthInPixels;
   DWORD               dwFrameHeightInLines;
   DWORD               nbFieldPerFrame;
   VIDEO_FIELD_DESC    FieldInfo[nbFieldPerFrame];
} VideoPropHeader;
```

Video Format Token

```
enum {FORMAT_UNKNOWN, FORMAT_PAL_SQUARE, FORMAT_PAL_CCIR_601,
   FORMAT_NTSC_SQUARE, FORMAT_NTSC_CCIR_601,...} VIDEO_FORMAT;
```

The format token indicates that a known standard is defined for the following data fields. Those fields must be filled, but their value can be expected to be the defined standard. If the format is defined as `FORMAT_UNKNOWN` then the fields may contain special values. Known tokens are defined in the table below:

Token	*Standard*	*Refresh Rate*	*H-Total*	*V-Total*	*Frame Aspect Ratio*	*Frame Width*	*Frame Height*	*Pixel Aspect Ratio (derived)*
NTSC CCIR 601	NTSC	60	858	525	0x0004 0003 (4:3)	720	485	2160:1940
NTSC SQUARE	NTSC	60	780	525	0x0004 0003 (4:3)	640	480	1:1
PAL CCIR 601	PAL	50	864	625	0x0004 0003 (4:3)	720	576	2160:2304
PAL SQUARE	PAL	50	944	625	0x0004 0003 (4:3)	768	576	1:1

Video Standard

```
enum {STANDARD_UNKNOWN, STANDARD_PAL, STANDARD_NTSC, STANDARD_SECAM}
   VIDEO_STANDARD;
```

Defines standards such as NTSC, PAL etc. Implicitly defines vertical refresh rate.

Vertical Refresh Rate

Used when an unknown standard is specified. Normally, 60 for NTSC, and 50 for PAL.

H-Total in T

Defines the horizontal total, in T (one luminance sample: pixel)

V-Total in Lines

Defines the vertical total, in lines.

Active Frame Aspect Ratio

The aspect ratio is stored as a DWORD value with a word each storing the x:y ratio. For example, 1 to 1 is 0x00010001. Standard values for television is 4:3 or 16:9. This value can be used with the frame width and height to calculate the pixel aspect ratio.

Active Frame Width in Pixels

Defines the active frame width in pixels. The bitmap might digitize a region that is smaller or bigger than the active video width.

Active Frame Height in Lines

Defines the frame height in lines. The bitmap might digitize a region that is smaller or bigger than the active video height.

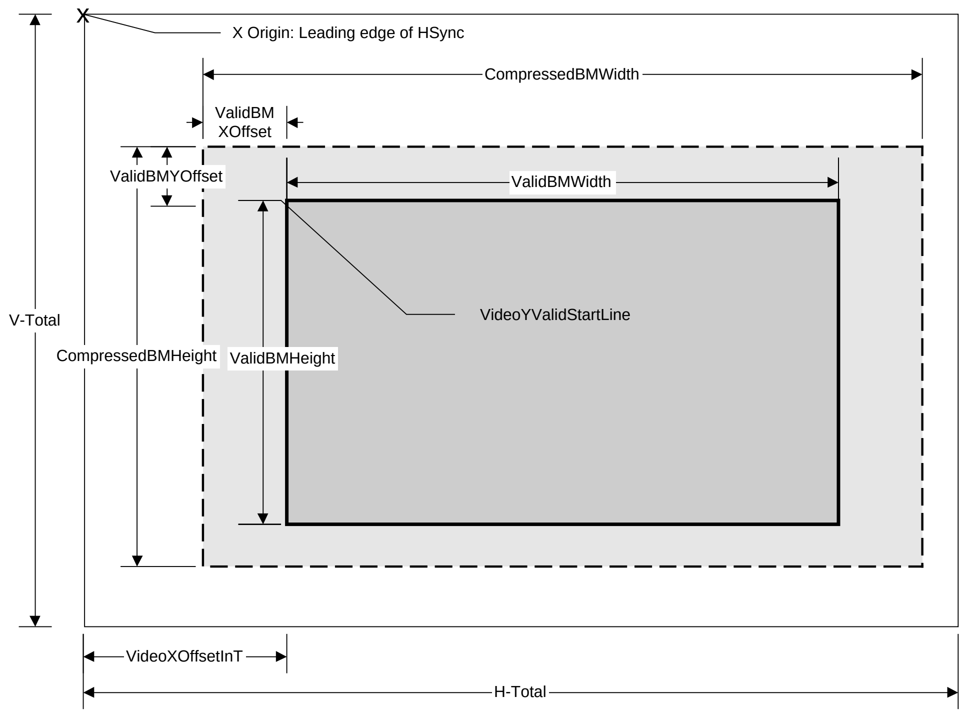
Number of Fields Per Frame

One or two, depending on whether the video is interlaced or progressive.

Field Framing Information

The field framing information defines where the compressed image is with respect to the video signal. The data is present for each field (and may be different).

```
Typedef struct {
   DWORD   CompressedBMHeight;
   DWORD   CompressedBMWidth;
   DWORD   ValidBMHeight;
   DWORD   ValidBMWidth;
   DWORD   ValidBMXOffset;
   DWORD   ValidBMYOffset;
   DWORD   VideoXOffsetInT;
   DWORD   VideoYValidStartLine;
} VIDEO_FIELD_DESC;
```



Compressed Bitmap Height and Width

The compressed bitmap height and width represent the size of the compressed image. For JPEG, these values are multiples of 8.

Valid Bitmap Height and Width, X and Y Offset

The valid bitmap height, width and x and y offsets represent the size of the valid data within the compressed bitmap. Because padding may be required when compressing, it is not guaranteed that all the data within the compressed image is valid. Note that compressing blanking is still valid. In the case where all the compressed bitmap comes from the video signal, then the valid height and width are equal to the compressed height and width, and the offsets are 0.

Valid X-Offset In T

The `VideoXOffsetInT` is used to locate the x position of the start of the valid bitmap with reference to the video signal. The value is a measurement in units of T, which is one luminance-sampling clock, from the leading edge of the horizontal sync pulse (CCIR 624-3).

Valid Y Start Line

The `VideoYValidStartLine` field is used to locate the line that the valid bitmap starts on. This value will be different for each field. (CCIR 624-3).

A typical value for the Framing Information for NTSC CCIR 601 would be:

Vide	*Co*	*Co*	*Vali*	*Vali*	*Vali*	*Va*	*Vide*	*Video*

o Type	*mpressed BM Height*	*mpressed BM Width*	*d BM Height*	*d BM Width*	*d BM X Offset*	*lid BM Y Offset*	*o X Offset In T*	*Y Valid Start Line*
Field 1	248	720	248	720	0	0	122	20
Field 1	248	720	248	720	0	0	122	283

Source and Timecode Information

The following information is used to describe timecode and source information inside an AVI file.

Base Timecode Structure

All SMPTE timecode information is stored in the following format:

```
typedef union _timecode {
   struct {
      WORD      wFrameRate;  // 0 is 30 drop.  do we need other drop
                             // frame rates?
      WORD      wFrameFract; // fractional frame. full scale is always 0x10000
      LONG      lFrame;
   };
   __int64  qwAll;
} TIMECODE, *PTIMECODE;
```

This format allows for a frame, and a frame fractional value to be specified, with the frame rate stored in the low word of the value. As the values are stored, two timecodes can be compared and subtracted. For drop frame code, no `lFrame` values are skipped, so a drop frame timecode of 1:00:00;00 would be a `lFrame` value of 107892.

Timecode Discontinuity Table (Stream) 'tcdl'

The following structure defines a timecode discontinuity table. This would be stored as a stream itself. The table would either be a table alone in the 'indx' chunk of that stream, or if the table is large, an index to the table could be stored with the table itself stored as a ##ix in the 'movi' data.

```
#define FILM_SEQUENCE_NONE       0
#define FILM_SEQUENCE_AABBBCCDDD 1
#define FILM_SEQUENCE_AAABBCCCDD 2
// structure of a timecode discontinuity table. note that this fits
// within the definition of the structure of an index chunk
//
typedef struct _timecode_dl_chunk {
   FOURCC   fcc;              // ’##ix’
   DWORD    cb;               // sizeof this structure less 8
   WORD     wLongsPerEntry;   // must be 4 or more
   BYTE     bIndexSubType;    // must be 0
   BYTE     bIndexType;       // AVI_INDEX_IS_DATA
   DWORD    nEntriesInUse;    // index of first unused entry in aIndex array
   DWORD    dwChunkId;        // ’tcdl’ (timecode discontinuity list)
   DWORD    dwReserved[3];    // future, must be 0
   struct _timecode_dl_entry {
      QUADWORD  qwTick;       // time in terms of this streams’s tick rate
      TIMECODE  timecode;     // timecode
      DWORD     dwUser;       // timecode user data
      struct {
         DWORD  ColorFrame : 4;       // Which frame in color sequence
         DWORD  ColorSequence : 4;    // Duration in frames of complete sequence
         DWORD  FilmFrame : 5;        // Offset into pull-down sequence
         DWORD  FilmSequenceType : 3; // One of FILM_SEQUENCE_XXX defines
         DWORD  Reserved : 16;        // Future use - set to 0
      } Flags;
      TCHAR     szReelId[32]; // source id
   } aIndex[ ];
};
```

The Flags structure can optionally specify color framing, and film pull-down information. If all fields are 0 then no such optional information is present. Notice that there is no explicit flag for drop frame NTSC since that information in already stored in the TIMECODE structure.

If `ColorSequence` is non-zero value, then `ColorFrame` contains a frame number from 0 to `ColorSequence-1`, which specifies the color framing of the frame specified in the timecode structure.

If `FilmSequenceType` is not `FILM_SEQUENCE_NONE`, then `FilmFrame` contains the offset into the pull-down sequence corresponding to the frame specified in the timecode structure. For example, if `FilmFrame` is 1 and `FilmSequenceType` is `FILM_SEQUENCE_AABBBCCDDD` then the referenced frame is a BB frame. Again, if `FilmFrame` is 1 and `FilmSequenceType` is `FILM_SEQUENCE_AAABBCCCDD` then the referenced frame is an AB frame.

Timecode Stream ‘time’

This structure defines a timecode value for every tick of the timecode stream

```
// structure of a timecode stream data chunk. for storing every
// event in a timecode stream rather than just the discontinuities
//
typedef struct _timecode_stream_chunk {
   FOURCC   fcc;             // ’##ix’
   DWORD    cb;              // sizeof this structure less 8
   WORD     wLongsPerEntry;  // must be 3
   BYTE     bIndexSubType;   // must be 0
   BYTE     bIndexType;      // AVI_INDEX_IS_DATA
   DWORD    nEntriesInUse;   // index of first unused entry in aIndex   array
   DWORD    dwChunkId;       // ’time’ (timecode stream)
   DWORD    dwReserved[3];   // future, must be 0
   struct _timecode_stream_entry {
      TIMECODE  timecode;    // timecode
      DWORD     dwUser;      // timecode user data
      struct {
         DWORD  ColorFrame : 4;       // Which frame in color sequence
         DWORD  ColorSequence : 4;    // Duration in frames of complete sequence
         DWORD  FilmFrame : 5;        // Offset into sequence
         DWORD  FilmSequenceType : 3; // One of FILM_SEQUENCE_XXX defines
         DWORD  Reserved : 16;        // Future use - set to 0
      } Flags;
   } aIndex[ ];
};
```

Film Transfer Log Information

```
#define FILM_LOG_EVERTZ          1
#define FILM_LOG_FLEX            2
// structure of a film transfer log.  Note that this fits
// within the definition of the structure of an index chunk
//
typedef struct _film_transfer_chunk {
   FOURCC   fcc;              // ’##ix’
   DWORD    cb;               // sizeof this structure less 8
   WORD     wLongsPerEntry;   // must be 4 or more
   BYTE     bIndexSubType;    // must be 0
   BYTE     bIndexType;       // AVI_INDEX_IS_DATA
   DWORD    nEntriesInUse;    // index of first unused entry in aIndex array
   DWORD    dwChunkId;        // ’film’ (film transfer log)
   DWORD    dwReserved[3];    // future, must be 0
   struct _film_transfer_header {
      DWORD     dwLogType;    // Which type of FILM_LOG_XXX file format
      DWORD     dwHeaderSize; // Size of this _film_transfer_header structure
      DWORD     dwReserved[4];// Future use - set to 0
      // DWORD  dwUserDefined[];  // Optional user data
   } FilmHeader;
   // BYTE  bTransferLog[];     // The actual transfer log begins here
};
```

The size of the `FilmHeader` is given by the `dwHeaderSize`. This allows for user defined additions to the header, which appear at the end of the `FilmHeader`. The ASCII film transfer log begins immediately after the `FilmHeader` structure. The size of the ASCII data can be determined from the `dwHeaderSize` of the header and the `cb` size of the entire structure. The ASCII data should be padded with NULLs to the nearest DWORD structure size.

The contents of the `bTransferLog` array is byte for byte the contents of the ASCII transfer log file of the type given in the `dwLogType` field. For now, two types are defined:

FILM_LOG_EVERTZ: the detailed .FTL file format can be obtained from

Evertz Microsystems, Ltd.

3465 Mainway

Burlington, Ontario L7M 1A9, Canada

905-335-3700

FILM_LOG_FLEX: the detailed FLEX file format can be obtained from

Time Logic, Inc.

11992 Challenger Ct.

Moorpark, CA 93021

805-529-1155

Microsoft-Defined Tombstone Data

The following tombstone data has already been defined by Microsoft:

Additional AVI information following the ‘strl’ chunk in the AVI header LIST ‘hdrl’ chunk. (from *AVI RIFF form, Video for Windows SDK*):

Additional Header Data

After the ‘strl’ chunk, some AVI files might have additional header data. Additional information can include timecode chunks that apply to the whole file. The following timecode four-character codes are useful in AVI files:[2]

Four-character code	**Description**
ISMP	Indicates the chunk contains the SMPTE timecode of the digitization starting point. The time is expressed as a zero-terminated text string of the form HH:MM:SS.FF. If performing MCI capture in AVICAP, this chunk is automatically set from the MCI start time.
IDIT	Indicates the chunk specifies the time and date digitizing commenced. This time is contained in an ASCII string of

[2] Note that these chunks are Ixxx chunks, and as such, should be found in an INFO chunk (see comment below)?

	exactly 26 characters and has the format "Wed Jan 02 02:03:55 1990\n\0". The **ctime** and **asctime** functions can be used to create strings in this format.

The 'ISMP' timecode chunk usually corresponds to the starting timecode copied from the first sample of a captured sequence. Use this chunk to index the captured sample to the original sample. For example, if you need to recapture a video sequence, you can use the timecode to reposition the original sequence and restart capture from that point.

The 'IDIT' timecode chunk usually corresponds to the time the sample was captured. This timecode provides a reference to the age of a sample and creates a history if a series of samples is captured.

The weakness with the above section is that the ISMP information does not include any reel ID, so the effective timecode is meaningless.

Additional information, global to RIFF files (from *Multimedia File Formats, Windows 3.1 SDK*):

The INFO List Chunk

The "INFO" list is a registered global form type that can store information that helps identify the contents of the chunk. This information is useful but does not affect the way a program interprets the file; examples are copyright information and comments. An "INFO" list is a "LIST" chunk with list type "INFO." The following example shows a sample "INFO" list chunk:

```
LIST('INFO'  INAM("Two Trees"Z)
          ICMT("A picture for the opening screen"Z) )
```

An "INFO" list should contain only the following chunks. New chunks may be defined, but an application should ignore any chunk it doesn't understand. The chunks listed below may only appear in an "INFO" list. Each chunk contains a ZSTR, or null-terminated text string.

Chunk ID	**Description**
IARL	*Archival Location*. Indicates where the subject of the file is archived.
IART	*Artist*. Lists the artist of the original subject of the file; for example, "Michaelangelo."
ICMS	*Commissioned*. Lists the name of the person or organization that commissioned the subject of the file; for example, "Pope Julian II."
ICMT	*Comments*. Provides general comments about the file or the subject of the file. If the comment is several sentences long, end each sentence with

	a period. Do not include new-line characters.
ICOP	*Copyright*. Records the copyright information for the file; for example, "Copyright Encyclopedia International 1991." If there are multiple copyrights, separate them by a semicolon followed by a space.
ICRD	*Creation date*. Specifies the date the subject of the file was created. List dates in year-month-day format, padding one-digit months and days with a zero on the left; for example, "1553-05-03" for May 3, 1553.
ICRP	*Cropped*. Describes whether an image has been cropped and, if so, how it was cropped; for example, "lower-right corner."
IDIM	*Dimensions*. Specifies the size of the original subject of the file; for example, "8.5 in h, 11 in w."
IDPI	*Dots Per Inch*. Stores dots per inch setting of the digitizer used to produce the file, such as "300."
IENG	*Engineer*. Stores the name of the engineer who worked on the file. If there are multiple engineers, separate the names by a semicolon and a blank; for example, "Smith, John; Adams, Joe."
IGNR	*Genre*. Describes the original work, such as "landscape," "portrait," "still life," etc.
IKEY	*Keywords*. Provides a list of keywords that refer to the file or subject of the file. Separate multiple keywords with a semicolon and a blank; for example, "Seattle; aerial view; scenery."
ILGT	*Lightness*. Describes the changes in lightness settings on the digitizer required to produce the file. Note that the format of this information depends on hardware used.
IMED	*Medium*. Describes the original subject of the file, such as "computer image," "drawing," "lithograph," and so on.
INAM	*Name*. Stores the title of the subject of the file, such as "Seattle From Above."
IPLT	*Palette Setting*. Specifies the number of colors requested when digitizing an image, such as "256."
IPRD	*Product*. Specifies the name of the title the file was originally intended for, such as "Encyclopedia of Pacific Northwest Geography."
ISBJ	*Subject*. Describes the contents of the file, such as "Aerial view of Seattle."
ISFT	*Software*. Identifies the name of the software package used to create the file, such as "Microsoft WaveEdit."

ISHP	*Sharpness*. Identifies the changes in sharpness for the digitizer required to produce the file (the format depends on the hardware used).
ISR C	*Source*. Identifies the name of the person or organization who supplied the original subject of the file; for example, “Trey Research.”
ISRF	*Source Form*. Identifies the original form of the material that was digitized, such as “slide,” “paper,” “map,” and so on. This is not necessarily the same as IMED.
ITC H	*Technician*. Identifies the technician who digitized the subject file; for example, “Smith, John.”

6.0 Other File Issues

Issues

Continuation Over Multiple Files

With restrictions to the maximum file format and limitations on the maximum size of disk partitions, it may be desirable to implement a mechanism that specifies a linkage across multiple files. It will certainly be desired that the data of one stream of video can be captured across multiple files. For the moment, this will be external to the AVI file.

Minimized Time to Open Files

tbd

Simultaneous Reading and Writing / Multiple Access

tbd

Issues

7.0 MJPG Motion JPEG DIB Extensions

Issues

ISO vs. Non ISO Formats

For interoperability between various hardware and software codecs, it is not sufficient that the file format be standardized; the contents of the video data chunks must be standardized. In order to comply with industry standards, the AVI MJPG specification and OMF 2.0 file format specifications, the standard ISO Motion JPEG specification is proposed, with certain additions to support efficient identification of the interleaved fields in the frame.

Non-ISO formats could be used, provided that they provide the same outward appearance as the ISO data chunk. This would require different codecs to decompress and compress the data.

8-bit vs. 10-bit YUV

This document currently addresses 8-bit 4:2:2 YUV as per the Microsoft specification.

Video vs. Bitmap

Issues between the fact that a bitmap is a pure rectangle, while video starts with half lines; Motion JPEG compresses on multiple of 8 lines, so what is the frame of video that is compressed vs. the real frame.

See the section of the Video Properties Header in Section 5.0.

Proposed Data Chunk Format

The data chunk format is defined as per the Microsoft *JPEG DIB FORMAT* Technical Note, Type 2: Motion JPEG DIB.

From the DIB spec:

Type 2: Motion JPEG

Motion JPEG DIBs shall accommodate interchange formats which satisfy the "General sequential and progressive syntax" (ISO 10918 Part 1, Annex B, Para. B.2). A set of images of this type with compatible parameters can be placed in an AVI file to describe a motion sequence. Frame headers for these DIBs shall be limited to those specified in Para B.2.2 of the cited Annex B. These types are SOF0, SOF1, SOF2, SOF3, SOF9, SOF10 and SOF11. Of the types accommodated, this specification provides implementation only for the Baseline Sequential DCT (SOF0).

This DIB type contains incomplete JPEG data (Abbreviated Format per ISO 10918) and is not intended for stand-alone single image frame disk files. It may be used within RIFF files and other contexts where it is appropriate to:

a. Decode an image without supplying the associated JPEG Huffman tables. This presumes the codec has been properly pre-initialized prior to image decode.

b. Request encoder output of compressed image data absent embedded Huffman Tables.

All motion JPEG data will use YCbCr encoding.

In an AVI sequence all JPEG frames will be key frames as this ensures that within the AVI and Video for Windows architecture all frames will be directly and independently addressable.

For optimal size and speed during playback of an AVI file the Huffman data used by motion JPEG will be fixed and defined by this document. This will make the individual frames of every motion sequence smaller and more efficient to play back. Also as all sequences of motion images use the same Huffman data and color space it is much more likely that motion data can be directly exchanged without re-compression. A definition of the Huffman data will be provided in MMREG.H (which is listed at the end of this document) as a byte string which can be concatenated onto the start of a motion JPEG image to form a valid still JPEG image-

```
  MJPGDHTSeg = { X’FF’, DHT, length, JPEG Huffman table
parameters }
```

Q-table data is present and may vary in every frame of a motion sequence to permit control over the bandwidth of sequences that contain bursts of frames of varying levels of complexity. The restart interval used during the compression process may also vary for every frame.

Only the interleaved form of YCrCb images is supported for motion JPEG data. This implies that only one SOS segment will be present in any particular motion JPEG image.

The following applies (again from MS DIB spec):

As in the JPEG DIB format the JPEG stream syntax is used for the image data with the following constraints. The following JPEG marker codes SOI, DRI, DQT, SOF0, SOS and EOI are expected (mandatory) in the image data chunk, and the constrained values shown in the example below are mandatory for the image data within the AVI stream.

Any parameters in the SOF0 (frame) and SOS (start of scan) headers that are duplicated in the `BITMAPINFOHEADER` for JPEG must be the same. This would include Sample Precision, subsampling, number of components (as implied by `JPEGColorSpaceID`), etc. The number of lines and samples per lines in the SOF0 segment and the width and height defined in the format chunk must match the main AVI header width and height values. All of these values are expected to remain the same for every image data chunk in the AVI sequence.

Within the image data chunk two JPEG segments beginning with the SOI marker and ending with the EOI marker are allowed to accommodate field-interleaved streams. There is an APP0 marker immediately following the SOI marker that contains information about the video image. Specifically, this allows the identification of the two fields that comprise an interleaved frame. This APP0 marker is expected to have the first 4 bytes following the length bytes set to the characters ’A’, ’V’, ’I’, ’1’. The next byte indicates which field the JPEG data was compressed from and has an expected value of 0x01 for the FIRST JPEG data segment and 0x02 for the SECOND segment, indicating the FIRST and SECOND fields respectively. If the stream is not field interleaved then this value will be 0x00 and there will only be one JPEG segment. The remaining seven bytes are expected to be set to 0 and will be ignored by the codec.

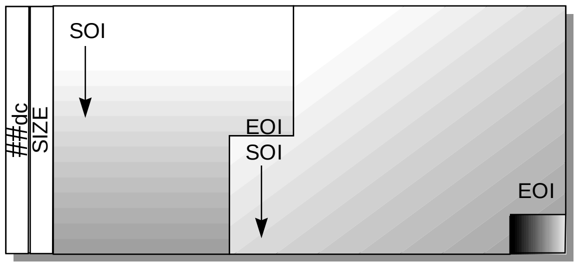
If a codec cannot handle the interleaved fields, the codec will use only the FIRST field and will replicate the lines as necessary to provide an image that conforms to the image size defined in the main AVI header. Conversely if a capture system only accesses a single field of each source frame only a single field image may be present in a JPEG stream. This implies that the single field data should be used as the source of both fields by a decompressor that wishes to process full-interlaced data.

And a new proposal:

The APP0 marker will have an added use beside the field polarity and will also specify the size of the current field (entirely, from SOI to EOI), as well as any padding required. This is to allow applications and codecs to avoid being forced to scan for the second SOI-EOI pair. The size information in the APP0 marker does not include any information about a second field in the ##dc chunk. This way, fields can be copied as integral units without having to have the contained data modified for the presence and size of neighbouring fields.

The byte following the polarity will be reserved for future expansion. The remaining 8 bytes of this marker will be used to store two sizes values. The first value specifies the length in bytes of the field (entirely, from SOI to EOI) including any padding after the EOI (which should be small) and the second value specifies the size of the same field excluding any padding after the EOI. Both size values will be stored in the usual ISO JPEG fashion where bytes are stored in decreasing order with the most significant byte being first. The length of this marker was previously 14 and is now set to 16.

MJPG DIB RIFF chunk (##dc)



An example MJPG frame with two interleaved fields

▨ First field (SOI EOI pair)

▨ Second field (SOI EOI pair)

■ Padding

MJPEG Baseline DCT - YCbCr DIB Map

A Baseline DCT - YCbCr will now therefore have the following look :

```
SOI  (xFFD8)
APP0 (xFFE0),
     Length (16),
     "AVI1", Polarity, 0, FieldSize,
FieldSizeLessPadding.
DRI  (xFFDD),
     Length (4),
     Restart interval.
DQT  (xFFDB),
     Length (132),
     Precision (0), Table ID (0),
     DQT data [64] for table 0,
     Precision (0), Table ID (1),
     DQT data [64] for table 1.
SOF0 (xFFC0),
     Length (17),
     Sample Precision (8),
     Number of lines (biHeight : multiple of 8),
     Sample per line (biWidth : multiple of 16),
     Number of components (3)
     ID of 1st component (1),
     Sampling ratio of 1st component (H,V),
     Q-table ID of 1st component (0),
     ID of 2nd component (2),
     Sampling ratio of 2nd component (H,V),
     Q table ID of 2nd component (1),
     ID of 3rd component (3),
     Sampling ratio of 3rd component (H,V),
     Q table ID of 3rd component (1).
SOS  (xFFDA),
     Length (12),
     Number of components (3)
     ID of 1st component (1),
     DC and AC Huffman ID for 1st component (0,0),
     ID of 2nd component (2),
     DC and AC Huffman ID for 2nd component (1,1),
     ID of 3rd component (3),
     DC and AC Huffman ID for 3rd component (1,1),
     Start of spectral (0),
     End of spectral (63),
     Successive approximation bit position, high and
low (0,0).
*** IMAGE DATA ***
There could be some Restart markers.
Values are from xFFD0 to xFFD7.
EOI (xFFD9).
```

Be sure to have a single DQT as the previous ones will be overridden.

The order of the markers may affect some codecs.

IDs for the 3 component as recommended by the Microsoft DIB specification are 1,2 and 3.

Sector Alignment and Padding

The ##dc chunks may be sector aligned if necessary. As such, a padding chunk would be inserted following the ##dc to preserve the RIFF format. Padding between fields should be discouraged; however the ISO specification allows padding if the data contains 0xFF for each padded byte.

8.0 OpenDML File Format Certification Procedures

Concurrently Develop a Test Suite to measure compatibility

Test older Codecs with new AVI files.

Test old AVI files with new Codecs.

Circulate all files through industry (i.e., everybody tests everybody else's files).

File checker routine to verify AVI format. Program may grow to include test decompressions using software Codec.

Test using MCIAVI with standard codecs.

Go to MS porting labs for specific video file format / Codec test week.