Adobe

Adobe Photoshop® 4.0

File Format Specification

Version 4.0.1 Release 1
April 1997

Most of the material for this document was derived from earlier works by Thomas Knoll, Mark Hamburg and Zalman Stern. Additional contributions came from David Corboy, Kevin Johnston, Sean Parent and Seetha Narayanan. It was then compiled and edited by Dave Wise and Paul Ferguson. It was later edited for content and updates by Andrew Coven.

Version History

Date	Author	Status
18 April1997	Andrew Coven	Extracted from SDK documentation for standalone document.

1. Introduction

Welcome to the Adobe Photoshop® File Format Specification!

This document is the detailed specification of the Adobe Photoshop File Format, and any other pertinent file formats that Adobe Photoshop reads and writes.

Audience

This toolkit is for C programmers who wish to write plug–ins for Adobe Photoshop on Macintosh and Windows systems.

This guide assumes you are proficient in the C programming language and its tools. The source code files in this toolkit are written for Metrowerks CodeWarrior on the Macintosh, and Microsoft Visual C++ on Windows.

You should have a working knowledge of Adobe Photoshop, and understand how plug–in modules work from a user's viewpoint. This guide assumes you understand Photoshop terminology such as *paths*, *layers* and *masks*. For more information, consult the *Adobe Photoshop User Guide*.

This guide does not contain information on creating plug–in modules for Unix versions of Photoshop. The Photoshop Unix SDK is available on the Photoshop Unix product CD. You must purchase the product CD to obtain the SDK.

About this guide

This programmer's guide is designed for readability on screen as well as in printed form. The page dimensions were chosen with this in mind. The Frutiger and Minion font families are used throughout the manual.

To print this manual from within Adobe Acrobat Reader, select the "Shrink to Fit" option in the Print dialog.

GAP SDK tech notes mailing list

The Adobe Developers Association maintains an area on Adobe's world-wide-web site: `http://www.adobe.com`, which includes the latest SDK public releases and technical notes. You can also have the technical notes e–mailed to you directly by joining the Graphics and Publishing SDK tech notes mailing list. The GAP SDK Tech Notes e-mail list is for Adobe After Effects, Adobe Illustrator, Adobe PageMaker, Adobe Photoshop, Adobe PhotoDeluxe, and Adobe Premiere. To receive tech notes, send an e-mail to:

```
gap-sdk-tn-requests@adobe.com
```

with the subject:

```
SUBSCRIBE
```

and these fields in your message body:

1. Your full name:
2. Business name:

3. Address:
4. City:
5. State:
6. Country:
7. Country code or Zip:
8. Area code and phone number (business is fine):
9. ADA member number:
   "`N/A`" if not a member; "`Info`" if want info.

SDK discussion mailing list

The Adobe Developers Association maintains an electronic mailing list that is used as peer discussion for developers. It is unmoderated and is populated by developers just like yourself, offering peer discussion of software development kit, Adobe plug-ins, and related issues. The mailing list is for discussion of all of the SDKs that fall under the ADA: Graphics and Publishing, which includes Adobe After Effects, Adobe Illustrator, Adobe PageMaker, Adobe Photoshop, Adobe PhotoDeluxe, and Adobe Premiere; Acrobat; FrameMaker; and PageMaker. To join the discussion send an e-mail to:

```
sdk-requests@adobe.com
```

with the subject:

```
SUBSCRIBE
```

and these fields in your message body:

1. Your full name:
2. Business name:
3. Address:
4. City:
5. State:
6. Country:
7. Country code or Zip:
8. Area code and phone number (business is fine):
9. ADA member number:
   "`N/A`" if not a member; "`Info`" if want info.

2. Document File Formats

Adobe Photoshop saves a user's document in one of several formats, which are listed under the pop–up menu in the **Save** dialog. This chapter documents these standard formats.

The formats discussed in this chapter include Photoshop 3.0 native format, Photoshop 4.0 additions to the 3.0 file format, Photoshop EPS format, Filmstrip format, and TIFF format.

For more information about file formats, you may wish to consult the *Encyclopedia of Graphics File Formats* by James D. Murray & William vanRyper (1994, O'Reilly & Associates, Inc., Sebastopol, CA, ISBN 1–56592–058–9).

Image resource blocks

Image resource blocks are the basic building unit of several file formats, including Photoshop's native file format, JPEG, and TIFF. Image resources are used to store non–pixel data associated with an image, such as pen tool paths. (They are referred to as resource data because they hold data that was stored in the Macintosh's resource fork in early versions of Photoshop.)

The basic structure of Image Resource Blocks is shown in table .

Table 2–1: Image resource block

Type	Name	Description
OSType	Type	Photoshop always uses its signature, `8BIM`.
int16	ID	Unique identifier (see table 10–2).
PString	Name	A pascal string, padded to make size even (a null name consists of two bytes of 0)
int32	Size	Actual size of resource data. This does not include the `Type`, `ID`, `Name`, or `Size` fields.
Variable	Data	Resource data, padded to make size even

Image resources use several standard ID numbers, as shown in table 2–2. Not all file formats use all ID's. Some information may be stored in other sections of the file.

Table 2–2: Image resource IDs

ID		Description
Hex	Dec	
0x03E8	1000	Obsolete—Photoshop 2.0 only. Contains five `int16` values: number of channels, rows, columns, depth, and mode.
0x03E9	1001	Optional. Macintosh print manager print info record.
0x03EB	1003	Obsolete—Photoshop 2.0 only. Contains the indexed color table.
0x03ED	1005	`ResolutionInfo` structure. See Appendix A.
0x03EE	1006	Names of the alpha channels as a series of Pascal strings.
0x03EF	1007	`DisplayInfo` structure. See Appendix A .
0x03F0	1008	Optional. The caption as a Pascal string.
0x03F1	1009	Border information. Contains a fixed-number for the border width, and an int16 for border units (1=inches, 2=cm, 3=points, 4=picas, 5=columns).
0x03F2	1010	Background color. See the Colors file information in chapter 9.
0x03F3	1011	Print flags. A series of one byte boolean values (see Page Setup dia-log): labels, crop marks, color bars, registration marks, negative, flip, interpolate, caption.
0x03F4	1012	Grayscale and multichannel halftoning information.
0x03F5	1013	Color halftoning information.
0x03F6	1014	Duotone halftoning information.
0x03F7	1015	Grayscale and multichannel transfer function.
0x03F8	1016	Color transfer functions.
0x03F9	1017	Duotone transfer functions.
0x03FA	1018	Duotone image information.
0x03FB	1019	Two bytes for the effective black and white values for the dot range.

Table 2–2: Image resource IDs (Continued)

ID		Description
Hex	**Dec**	
0x03FC	1020	Obsolete.
0x03FD	1021	EPS options.
0x03FE	1022	Quick Mask information. 2 bytes containing Quick Mask channel ID, 1 byte boolean indicating whether the mask was initially empty.
0x03FF	1023	Obsolete.
0x0400	1024	Layer state information. 2 bytes containing the index of target layer. 0=bottom layer.
0x0401	1025	Working path (not saved). See path resource format later in this chapter.
0x0402	1026	Layers group information. 2 bytes per layer containing a group ID for the dragging groups. Layers in a group have the same group ID.
0x0403	1027	Obsolete.
0x0404	1028	IPTC-NAA record. This contains the **File Info...** information.
0x0405	1029	Image mode for raw format files.
0x0406	1030	JPEG quality. Private.
0x0408	1032	*New since version 4.0 of Adobe Photoshop:* Grid and guides information. See grid and guides resource format later in this chapter. *New since version 4.0 of Adobe Photoshop.*
0x040A	1034	*New since version 4.0 of Adobe Photoshop:* Copyright flag. Boolean indicating whether image is copyrighted. Can be set via Property suite or by user in **File Info...**
0x040B	1035	*New since version 4.0 of Adobe Photoshop:* URL. Handle of a text string with uniform resource locator. Can be set via Property suite or by user in **File Info...**
0x07D0-0x0BB6	2000-2998	Path Information (saved paths). See path resource format later in this chapter.
0x0BB7	2999	Name of clipping path. See path resource format later in this chapter.
0x2710	10000	Print flags information. 2 bytes version (=1), 1 byte center crop marks, 1 byte (=0), 4 bytes bleed width value, 2 bytes bleed width scale.

Grid and guides resource format

Adobe Photoshop 4.0 and later stores grid and guides information an image in an image resource block. These resource blocks consist of an initial 12 byte grid and guide header, which is always present, followed by a 5 byte blocks of specific guide information for guide direction and location, which are present if there are guides (`fGuideCount` > 0).

Table 2–3: Grid and guide header

Type	Name	Description
int32	fVersion	=1 for Photoshop 4.0.
VPoint	fGridCycle	Future implementation of document-specific grids. Initially, set the grid cycle to every quarter inch. At 72 dpi, that would be 18 * 32 = 576 (0x240).
int32	fGuideCount	Can be 0. Otherwise, number of guide resource blocks.

Table 2–4: Guide resource block

Type	Name	Description
int32	fLocation	Location of guide in document coordinates. Since the guide is either vertical or horizontal, this only has to be one component of the coordinate.
VHSelect	fDirection	Direction of guide. VHSelect is a system type of unsigned char where 0 = vertical, 1 = horizontal.

Grid and guide information may be modified using the Property suite. See the Callbacks chapter for more information.

Path resource format

Photoshop stores the paths saved with an image in an image resource block. These resource blocks consist of a series of 26 byte path point records, and so the resource length should always be a multiple of 26.

Photoshop stores its paths as resources of type `8BIM` with IDs in the range 2000 through 2999. These numbers should be reserved for Photoshop. The name of the resource is the name given to the path when it was saved.

If the file contains a resource of type `8BIM` with an ID of 2999, then this resource contains a Pascal–style string containing the name of the clipping path to use with this image when saving it as an EPS file.

The path format returned by `GetProperty()` call is identical to what is described below. Refer to the `IllustratorExport` sample plug–in code to see how this resource data is constructed.

Path points

All points used in defining a path are stored in eight bytes as a pair of 32–bit components, vertical component first.

The two components are signed, fixed point numbers with 8 bits before the binary point and 24 bits after the binary point. Three guard bits are reserved in the points to eliminate most concerns over arithmetic overflow. Hence, the range for each component is `0xF0000000` to `0x0FFFFFFF` representing a range of –16 to 16. The lower bound is included, but not the upper bound.

This limited range is used because the points are expressed relative to the image size. The vertical component is given with respect to the image height, and the horizontal component is given with respect to the image width. `[0,0]` represents the top–left corner of the image; `[1,1]` (`[0x01000000,0x01000000]`) represents the bottom–right.

In Windows, the byte order of the path point components are reversed; you should swap the bytes when accessing each 32–bit value.

Path records

The data in a path resource consists of one or more 26-byte records. The first two bytes of each record is a selector to indicate what kind of path it is. For Windows, you should swap the bytes before accessing it as a short (`int16`).

Table 2–5: Path data record types

Selector	Description
0	Closed subpath length record
1	Closed subpath Bezier knot, linked
2	Closed subpath Bezier knot, unlinked
3	Open subpath length record
4	Open subpath Bezier knot, linked
5	Open subpath Bezier knot, unlinked
6	Path fill rule record
7	Clipboard record

The first 26-byte path record contains a selector value of 6, path fill rule record. The remaining 24 bytes of the first record are zeroes. Paths use even/

odd ruling. Subpath length records, selector value 0 or 3, contain the number of Bezier knot records in bytes 2 and 3. The remaining 22 bytes are unused, and should be zeroes. Each length record is then immediately followed by the Bezier knot records describing the knots of the subpath.

In Bezier knot records, the 24 bytes following the selector field contain three path points (described above) for:

1. the control point for the Bezier segment preceding the knot,
2. the anchor point for the knot, and
3. the control point for the Bezier segment leaving the knot.

Linked knots have their control points linked. Editing one point modifies the other to preserve collinearity. Knots should only be marked as having linked controls if their control points are collinear with their anchor. The control points on unlinked knots are independent of each other. Refer to the *Adobe Photoshop User Guide* for more information.

Clipboard records, `selector=7`, contain four fixed-point numbers for the bounding rectangle (top, left, bottom, right), and a single fixed-point number indicating the resolution.

Photoshop 3.0 files

This is the native file format for Adobe Photoshop 3.0. It supports storing all layer information.

Table 2–6: Photoshop 3.0 file types

OS	Filetype/extension
Mac OS	`8BPS`
Windows	`.PSD`

Photoshop 3.0 files under Windows

All data is stored in big endian byte order; under Windows you must byte swap short and long integers when reading or writing.

Photoshop 3.0 files under Mac OS

For cross–platform compatibility, all information needed by Adobe Photoshop 3.0 is stored in the data fork. For interoperability with other Macintosh applications, however, some information is duplicated in resources stored in the resource fork of the file.

For compatibility with image cataloging applications, the `pnot` resource id 0 contains references to thumbnail, keywords, and caption information stored in other resources. The thumbnail picture is stored in a `'PICT'` resource, the keywords are stored in `'STR#'` resource 128 and the caption text is stored in `'TEXT'` resource 128. For more information on the format of these resources see *Inside Macintosh: QuickTime Components* and the *Extensis Fetch Awareness Developer's Toolkit.*

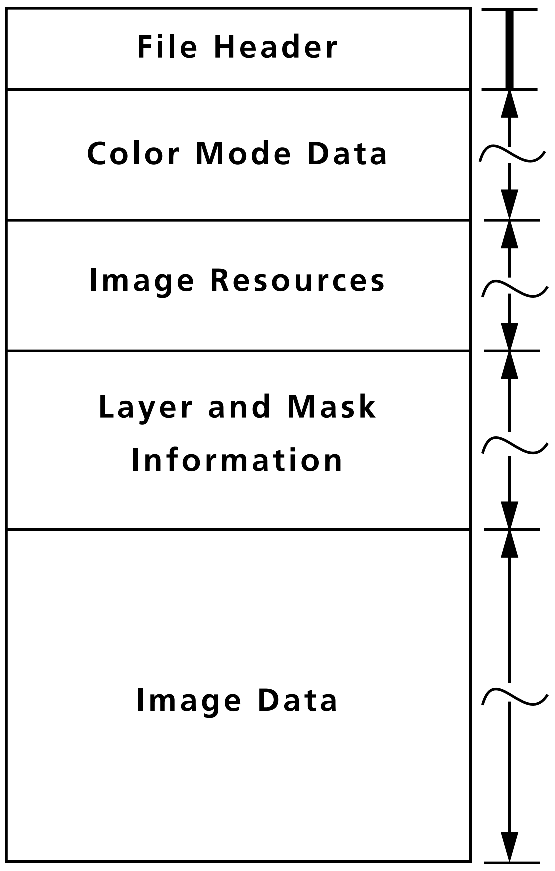
Photoshop also creates `'icl8'` –16455 and `'ICN#'` –16455 resources containing thumbnail images which will be shown in the Mac OS Finder.

All of the data from Photoshop's File Info dialog is stored in `'ANPA'` resource 10000. The data in this resource is stored as an IPTC–NAA record 2 and should be readable by various tools from Iron Mike. For more information on the format of this resource contact:

IPTC–NAA Digital Newsphoto Parameter Record
Newspaper Association of America
The Newspaper Center
11600 Sunrise Valley Drive
Reston VA 20091

Photoshop 3.0 file format

The file format for Photoshop 3.0 is divided into five major parts.



The file header is fixed length, the other four sections are variable in length.

When writing one of these sections, you should write all fields in the section, as Photoshop may try to read the entire section. Whenever writing a file and skipping bytes, you should explicitly write zeros for the skipped fields.

When reading one of the length delimited sections, use the length field to decide when you should stop reading. In most cases, the length field indicates the number of bytes, not records, following.

File header section

The file header contains the basic properties of the image.

Table 2–7: File header

Length	Name	Description
4 bytes	Signature	Always equal to 8BPS. Do not try to read the file if the signature does not match this value.
2 bytes	Version	Always equal to 1. Do not try to read the file if the version does not match this value.
6 bytes	Reserved	Must be zero.
2 bytes	Channels	The number of channels in the image, including any alpha channels. Supported range is 1 to 24.
4 bytes	Rows	The height of the image in pixels. Supported range is 1 to 30,000.
4 bytes	Columns	The width of the image in pixels. Supported range is 1 to 30,000.
2 bytes	Depth	The number of bits per channel. Supported values are 1, 8, and 16.
2 bytes	Mode	The color mode of the file. Supported values are: Bitmap=0; Grayscale=1; Indexed=2; RGB=3; CMYK=4; Multichannel=7; Duotone=8; Lab=9.

Color mode data section

Only indexed color and duotone have color mode data. For all other modes, this section is just 4 bytes: the length field, which is set to zero.

For indexed color images, the length will be equal to 768, and the color data will contain the color table for the image, in non–interleaved order.

For duotone images, the color data will contain the duotone specification, the format of which is not documented. Other applications that read Photoshop files can treat a duotone image as a grayscale image, and just preserve the contents of the duotone information when reading and writing the file.

Table 2–8: Color mode data

Length	Name	Description
4 bytes	Length	The length of the following color data.
Variable	Color data	The color data.

Image resources section

The third section of the file contains image resources. As with the color mode data, the section is indicated by a length field followed by the data. The image resources in this data area are described in detail earlier in this chapter.

Table 2–9: Image resources

Length	Name	Description
4 bytes	Length	Length of image resource section.
Variable	Resources	Image resources.

Layer and mask information section

The fourth section contains information about Photoshop 3.0 layers and masks. The formats of these records are discussed later in this chapter. If there are no layers or masks, this section is just 4 bytes: the length field, which is set to zero.

Table 2–10: Layer and mask information

Length	Name	Description
4 bytes	Length	Length of the miscellaneous information section.
Variable	Layers	Layer info. See table 2–12.
Variable	Global layer mask	Global layer mask info. See table 2–19.

Image data section

The image pixel data is the last section of a Photoshop 3.0 file. Image data is stored in planar order, first all the red data, then all the green data, etc. Each plane is stored in scanline order, with no pad bytes.

If the compression code is 0, the image data is just the raw image data.

If the compression code is 1, the image data starts with the byte counts for all the scan lines (rows * channels), with each count stored as a two–byte value. The RLE compressed data follows, with each scan line compressed separately. The RLE compression is the same compression algorithm used by the Macintosh ROM routine `PackBits`, and the TIFF standard.

Table 2–11: Image data

Length	Name	Description
2 bytes	Compression	Compression method. Raw data = 0, RLE compressed = 1.
Variable	Data	The image data. Planar order = RRR GGG BBB, etc.

Layer and mask records

Information about each layer and mask in a document is stored in the fourth section of the file. The complete, merged image data is not stored here; it resides in the last section of the file.

The first part of this section of the file contains layer information, which is divided into layer structures and layer pixel data, as shown in table 2–12. The second part of this section contains layer mask data, which is described in table 2–19.

Table 2–12: Layer info section

Length	Name	Description
4 bytes	Length	Length of the layers info section, rounded up to a multiple of 2.
Variable	Layers structure	Data about each layer in the document. See table 2–13.
Variable	Pixel data	Channel image data for each channel in the order listed in the layers structure section. See table 2–18.

Table 2–13: Layer structure

Length	Name	Description
2 bytes	Count	Number of layers. If <0, then number of layers is absolute value, and the first alpha channel contains the transparency data for the merged result.
Variable	Layer	Information about each layer (table 2–18).

Table 2–14: Layer records

Length	Name	Description
4 bytes	Layer top	The rectangle containing the contents of the layer.
4 bytes	Layer left	
4 bytes	Layer bottom	
4 bytes	Layer right	
2 bytes	Number channels	The number of channels in the layer.
Variable	Channel length info	Channel information. This contains a six byte record for each channel. See table 2–15.
4 bytes	Blend mode signature	Always `8BIM`.
4 bytes	Blend mode key	`'norm'` = normal `'dark'` = darken `'lite'` = lighten `'hue '` = hue `'sat '` = saturation `'colr'` = color `'lum '` = luminosity `'mul '` = multiply `'scrn'` = screen `'diss'` = dissolve `'over'` = overlay `'hLit'` = hard light `'sLit'` = soft light `'diff'` = difference
1 byte	Opacity	0 = transparent ... 255 = opaque
1 byte	Clipping	0 = base, 1 = non–base

Table 2–14: Layer records (Continued)

Length	Name	Description
1 byte	Flags	bit 0 = transparency protected bit 1 = visible
1 byte	(filler)	(zero)
4 bytes	Extra data size	Length of the extra data field. This is the total length of the next five fields.
24 bytes, or 4 bytes if no layer mask.	Layer mask data	See table 2–16.
Variable	Layer blending ranges	See table 2–17.
Variable	Layer name	Pascal string, padded to a multiple of 4 bytes.
These fields are new since version 4.0 of Adobe Photoshop:		
Variable	Adjustment layer info	See table 2–20.

Table 2–15: Channel length info

Length	Name	Description
2 bytes	Channel ID	0 = red, 1 = green, etc. –1 = transparency mask –2 = user supplied layer mask
4 bytes	Length	Length of following channel data.

Table 2–16: Layer mask / adjustment layer data

Length	Name	Description
4 bytes	Size	Size of the data. This will be either 0x14, or zero (in which case the following fields are not present).
4 bytes	Top	Rectangle enclosing layer mask.
4 bytes	Left	
4 bytes	Bottom	
4 bytes	Right	
1 byte	Default color	0 or 255
1 byte	Flags	bit 0 = position relative to layer bit 1 = layer mask disabled bit 2 = invert layer mask when blending
2 bytes	Padding	Zeros

Table 2–17: Layer blending ranges data

Length	Name	Description
4 bytes	Length	Length of layer blending ranges data
4 bytes	Composite gray blend source	Contains 2 black values followed by 2 white values. Present but irrelevant for Lab & Grayscale.
4 bytes	Composite gray blend destination	Destination Range
4 bytes	First channel source range	First channel source

Table 2–17: Layer blending ranges data (Continued)

Length	Name	Description
4 bytes	First channel destination range	First channel destination
4 bytes	Second channel source range	Second channel source
4 bytes	Second channel destination range	Second channel destination
...	...	...
4 bytes	Nth channel source range	Nth channel source
4 bytes	Nth channel destination ra nge	Nth channel destination

Table 2–18: Channel image data

Length	Name	Description
2 bytes	Compression	0 = Raw Data, 1 = RLE compressed.
Variable	Image data	If the compression code is 0, the image data is just the raw image data calculated as `((LayerBottom-LayerTop)*(LayerRight-LayerLeft))`. If the compression code is 1, the image data starts with the byte counts for all the scan lines in the channel `(LayerBottom-LayerTop)`, with each count stored as a two–byte value. The RLE compressed data follows, with each scan line compressed separately. The RLE compression is the same compression algorithm used by the Macintosh ROM routine PackBits, and the TIFF standard. If the Layer's Size, and therefore the data, is odd, a pad byte will be inserted at the end of the row. *New since version 4.0 of Adobe Photoshop:* If the layer is an adjustment layer, the channel data is undefined (probably all white.)

Table 2–19: Global layer mask info

Length	Name	Description
4 bytes	Length	Length of global layer mask info section.
2 bytes	Overlay color space	Overlay color space (undocumented).
8 bytes	Color components	4 * 2 byte color components
2 bytes	Opacity	0 = transparent, 100 = opaque.
1 byte	Kind	0=Color selected—i.e. inverted; 1=Color protected;128=use value stored per layer. This value is preferred. The others are for backward compatibility with beta versions.
1 byte	(filler)	(zero)

Photoshop 4.0 file format

The Photoshop 4.0 file format is an extension of the Photoshop 3.0 file format. It is essentially the same, with some additional image resources and resource blocks. Additional resources for 4.0 are:

1. Copyright flag. Image resource ID 1034 (0x040A). See table 2–2
2. URL ID. Image resource ID 1035 (0x040B). See table 2–2.
3. Grid and guides information. See table 2–2 and grid and guides resource format, earlier in this chapter.
4. Adjustment layers. See table 2–14 and 2–20.

Table 2–20: Adjustment layer info

Length	Name	Description
4 bytes	signature	always `'8BIM'`
4 bytes	key	OSType key for which adjustment type to use: `'levl'`=Levels `'curv'`=Curves `'brit'`=Brightness/contrast `'blnc'`=Color balance `'hue '`=Hue/saturation `'selc'`=Selective color `'thrs'`=Threshold `'nvrt'`=Invert `'post'`=Posterize
4 bytes	length	Length of adjustment data, below.
Variable	data	Adjustment data. Same as load file formats for each format. See Load File Formats chapter for information.

Photoshop EPS files

Photoshop 3.0 and later writes a high–resolution bounding box comment to the EPS file immediately following the traditional EPS bounding box comment. The comment begins with “%%HiResBoundingBox” and is followed by four numbers identical to those given for the bounding box except that they can have fractional components (i.e., a decimal point and digits after it). The traditional bounding box is written as the rounded version of the high resolution bounding box for compatibility.

Photoshop writes its image resources out to a block of data stored as follows:

```
%BeginPhotoshop: <length> <hex data>
```

Table 2–21: EPS parameters for BeginPhotoshop

Field	Definition
length	Length of the image resource data.
hex data	Image resource data in hexadecimal.

Photoshop includes a comment in the EPS files it writes so that it is able to read them back in again. Third party programs that write pixel–based EPS files may want to include this comment in their EPS files, so Photoshop can read their files.

The comment must follow immediately after the %% comment block at the start of the file. The comment is:

```
%ImageData: <columns> <rows> <depth> <mode> <pad channels> <block size>
<binary/hex> "<data start>"
```

Table 2–22: EPS parameters for ImageData

Field	Definition
columns	Width of the image in pixels.
rows	Height of the image in pixels.
depth	Number of bits per channel. Must be 1 or 8.
mode	Image mode. Bitmap/grayscale=1; Lab=2; RGB=3; CMYK=4.
pad channels	Number of other channels store in the file. Ignored when reading. Photo-shop uses this to include a grayscale image that is printed on non-color PostScript printers.
block size	Number of bytes per row per channel. Will be either 1 or formula (below): 1=Data is interleaved. `(columns*depth+7)/8`=Data is stored in line-interleaved format, or there is only one channel.
binary/ascii	1=Data is in binary format. 2=Data is in hex ascii format.
data start	Entire PostScript line immediately preceding the image data. This entire line should not occur elsewhere in the PostScript header code, butit may occur at part of a line.

Filmstrip files

Adobe Premiere 2.0 and later supports the filmstrip file format. Premiere users can export any video clip as a filmstrip. Refer to the *Adobe Premiere User Guide* for more information.

Adobe Photoshop 3.0 supports the filmstrip file type to allow each frame to be individually painted. The filmstrip file format is fairly simple, and is described in this section.

A filmstrip consists of a sequence of equal sized 32–bit images, known as frames. The channel order in the file is Red, Green, Blue, Alpha.

After each frame is an arbitrarily sized leader area, in which any type of information may be embedded. Adobe Premiere puts the timecode and frame number for the frame in this area. This area is ignored by Photoshop when the file is read.

Following all the frames is a 16 row trailer frame (it has the same width as the other frames). Adobe Premiere writes a yellow and black diagonal pattern in this area. The lower right corner of this area is actually an information record that exists at the very end of the file. This record is located by seeking to the end of the file minus the size of the record, then reading the record and verifying the signature field that it contains.

```
// Definition for filmstrip info record

typedef struct {
    long                signature;   // 'Rand'
    long                numFrames;   // number of frames in file
    short               packing;     // packing method
    short               reserved;    // reserved, should be 0
    short               width;       // image width
    short               height;      // image height
    short               leading;     // horiz gap between frames
    short               framesPerSec;// frame rate
    char                spare[16];   // some spare data.
} FilmStripRec, **FilmStripHand;
```

Table 2–23: FilmStripRec structure

Type	Field	Description
long	signature	This field must be set to the code Rand and is used to verify the validity of the record.
long	numFrames	This is the total number of frames in the file.
short	packing	This is the packing method used, currently only a value of 0 is defined, for no packing.
short	width	The width of each image, in pixels.
short	height	The height of each image, in pixels.
short	leading	The height of the leading areas, in pixels.
short	framesPerSec	The rate at which the frames should be played.

To locate the filmstrip info record, seek to the end of the file minus (`sizeof(FilmStripRec)`), then read in the FilmStrip record. Check the signature field for the code Rand to test for validity.

To locate the data for a particular frame, seek to

```
(frame * width * (height+leading) * 4)
```

then read the number of bytes in

```
(width * height * 4).
```

If the data is being placed into a Mac OS GWorld, the channels must be re–arranged from Red–Green–Blue–Alpha to Alpha–Red–Green–Blue.

To write a FilmStrip file, write each frame sequentially into the file, including the leading areas.

Then write this block of bytes:

```
((width * (height+leading) * 4) – sizeof(FilmStripRec)).
```

Finally, fill in and write the FilmStrip record to the file.

Note: The packing field should currently be zero. In the future packing methods may be defined for filmstrips, so any software which reads filmstrips should examine this field before opening the file.

TIFF files

The same image resources information found in Photoshop 3.0 files are stored in TIFF files under tag number 34377 (see Image Resource Blocks and Image Resources earlier in this chapter).

For TIFF files the caption data is stored in an image description tag 270 and all the information is stored as an IPTC–NAA record 2 in tag 33723. The tag number was chosen by inspecting files written by Iron Mike software, and is supposed to be defined in a Rich TIFF specification. The tag is also specified in:

NSK TIFF
    The Japan Newspaper Publishers & Editors Association
    Nippon Press Center Building
    2–2–1 Uchlsaiwai–cho
    Chiyoda–ku, Tokyo 100

For more information about the TIFF format see:

TIFF Revision 6.0
    `http: //www.adobe.com/supportservice/devrelations/resources.html#tiff`

In reading the files, the following order is used with information read lower on the list replacing information read higher:
    Image Description Tag (TIFF only)
    IPTC–NAA Tag (TIFF only)

Note: It is a bug that the TIFF information comes prior to the image resource information on this list. This means that an edit to the TIFF info will not be recognized unless the image resource information is removed. The TIFF data may be moved to after the image resource information in a future version of Photoshop.

Table 2–24 describes the standard TIFF tags and tag values that Photoshop 3.0 and later is able to read and write.

TIFF files under the Mac OS

For cross–platform compatibility, all TIFF information is stored in the data fork. For interoperability with other Macintosh applications, however, some information is duplicated in resources stored in the resource fork of the file.

For compatibility with image cataloging applications, the pnot resource id 0 contains references to thumbnail, keywords, and caption information stored in other resources. The thumbnail picture is stored in a 'PICT' resource, the keywords are stored in 'STR#' resource 128 and the caption text is stored in 'TEXT' resource 128. For more information on the format of these resources see *Inside Macintosh: QuickTime Components* and the *Extensis Fetch Awareness Developer's Toolkit.*

All of the data from Photoshop's File Info dialog is stored in 'ANPA' resource 10000. The TIFF file also contains 'STR ' resource -16396 indicating the application that created the TIFF file. The string is "Adobe Photoshop™ 3.0" for Photoshop 3.0 and "Adobe Photoshop® 4.0" for Photoshop 4.0.

Photoshop also creates 'icl8' –16455 and 'ICN#' –16455 resources containing thumbnail images which will be shown in the Mac OS Finder.

Table 2–24: TIFF Tags

Tag	Photoshop reads	Photoshop writes
IFD	First IFD in file	Only one IFD per file
NewSubFileType	Ignored	0
ImageWidth	1 to 30000	1 to 30000
ImageLength	1 to 30000	1 to 30000
BitsPerSample	1, 2, 4, 8, 16 (all same)	1, 8, 16
Compression	1, 2, 5, 32773	1, 5
PhotometricInterpretation	0, 1, 2, 3, 5, 8	0 (1–bit), 1 (8–bit), 2, 3,5,8
FillOrder	1	No
ImageDescription	Printing Caption	Printing Caption
StripOffsets	Yes	Yes
SamplesPerPixel	1 to 24	1 to 24
RowsPerStrip	Any	Single strip if not compressed, multiple strips if compressed.
StripByteCounts	Required if compressed	Yes
XResolution	Yes	Yes
YResolution	Ignored (square pixels assumed)	Yes
PlanarConfiguration	1 or 2	1
ResolutionUnit	2 or 3	2
Predictor	1 or 2	1 or 2
ColorMap	Yes	Yes
TileWidth	Yes	No
TileLength	Yes	No
TileOffsets	Yes	No
TileByteCounts	Required if compressed	No
InkSet	1	No
DotRange	Yes, if CMYK	Yes
ExtraSamples	Ignored (except for count)	0

Adobe

Adobe Photoshop™ 3.0

PLUG-IN TOOLKIT

Adobe Photoshop 3.0 Plug-in Toolkit

Most of the material for this document was derived from earlier works by Thomas Knoll, Mark Hamburg and Zalman Stern. Additional contributions came from David Corboy, Kevin Johnston, Sean Parent and Seetha Narayanan. It was then compiled and edited by Dave Wise.

Version History:

11/07/94	David J. Wise	First Draft
1/15/95	David J. Wise	First Release
2/08/95	Seetharaman Narayanan	MS-Windows Mods.

Introduction

How to Use This Toolkit

The Adobe Photoshop Plug-In Toolkit is for developers who wish to write their own plug-in modules for use with Adobe Photoshop. Photoshop plug-ins are called by Photoshop to perform specific functions, such as acquiring an image or filtering a portion of an image.

This toolkit documentation starts with information that is common to all the plug-in types. The rest of the document is broken up into chapters specific to each type of plug-in, or to special types of files.

The best way to use this toolkit documentation is to read this Introduction chapter, then read the chapter specific to the type of plug-in you're writing. You should then study and understand the sample plug-ins of the type you're writing.

Plug-In Overview

Adobe Photoshop plug-ins are separate files that contain code which allows either Adobe Systems, Inc. or third-party developers to extend Adobe Photoshop, without actually modifying the base application. Adobe Photoshop version 3.0 supports five kinds of plug-in modules:

1. Acquisition modules, which open an image in a new window. Acquisition modules can be used to interface to scanners or frame grabbers, read images in unsupported or compressed file formats, or to generate synthetic images. These modules are accessed through the Acquire sub-menu.
2. Export modules, which output an existing image. Export modules can be used to print to printers that do not have chooser-level driver support, or to save images in unsupported or compressed file formats. These modules are accessed through the Export sub-menu.
3. Filter modules, which modify a selected area of an existing image. These modules are inserted into the Filter menu.
4. File format modules which provide support for additional image formats. These appear in the format pop-up in the Open..., Save As... and Save a Copy... dialogs.
5. Parser modules...TBD

A quick word about types is in order here. The interface files talk in terms of **int32**'s and **int16**'s rather than longs and shorts when they specify 32-bit integers. **VRect**'s are like Macintosh **Rect**'s but they have 32-bit coordinates. All of these types are defined in **PITypes.h**.

Historical Note

The concept of plug-in modules has become popular among Macintosh application developers. Perhaps the best known example is Apple's HyperCard, with its support for XCMD's. One of the first companies to incorporate plug-in modules into their products was Silicon Beach, in its Digital Darkroom and SuperPaint products.

Silicon Beach's implementation of plug-in modules was well designed. Its good features include allowing the plug-in modules to reside in individual files (rather than having to be pasted into the application using ResEdit), allowing the plug-in modules to be placed anywhere (not just in the system folder), and allowing for future extensions by means of a version number.
Adobe Photoshop's implementation of plug-in modules is similar to that used by Silicon Beach. Its uses a similar calling sequence, and the same version number scheme.
Unfortunately, the detailed interface for Adobe Photoshop's plug-in modules is completely different from that used by Silicon Beach. The differences were required primarily to support color images and Adobe Photoshop's virtual memory scheme.

Installation

Macintosh

To install a plug-in module, all the user must do is drag the module's icon to one of the following folders: the same folder as the application or the plug-ins folder designated in the user's Photoshop preferences file. Photoshop 3.0 searches for plug-ins in the application folder, and throughout the tree of folders underneath the designated plug-ins folder. Aliases are followed during the search process. (Folders with names beginning with "¬" are ignored.)

Windows

To install a plug-in module, the user must copy the plug-in into the directory referred to in the PHOTOSHO.INI file with the profile string PLUGINDIRECTORY.
When Adobe Photoshop starts executing, it searches the files in the PLUGINDIRECTORY, looking for plug-in modules. When it finds a plug-in, it checks its version number, and if the version is supported, it adds the plug-in's name to the appropriate menu or to the list of extensions to be executed.
Each kind of plug-in module has its own 4-byte resource-type. For example, acquisition modules have the code '8BAM' (Note: the actual resource-type must be specified as _8BAM in your resource files to avoid a syntax error caused by the first character being a number). Adobe Photoshop searches for acquisition modules by examining the resources of all files in the PLUGINDIRECTORY that have file extension .8B*, for resources of type _8BAM. The nameID, the integer value which uniquely identifies the resource, for each 8BAM in the file must be consecutively numbered starting at 1.

Resources

'PiPL's

Definition

A Plug-In Property List, often called a 'PiPL' (pronounced "pipple") after its resource type code, is a flexible, extensible mechanism for representing plug-in metadata. This includes all information Photoshop needs to identify and load the plug-in as well as flags and other static properties that control the operation of the plug-in.

Plug-in Property Lists replace the Plug-in Module Information structure, often called a 'PiMI' (pronounced "pimmy") after its resource type code. A 'PiMI' is a fixed format record which originally contained only a version number. With the evolution of Photoshop's plug-in interface, this record expanded to include other information. The addition of multiple plug-in types resulted in the PiMI becoming a variant record with generic data at the beginning and a type specific data at the end. Further plug-in interface evolution required more complex metadata, such as an array of allowable file types for file format plug-ins.
The combination of variant and variable sized fields in the 'PiMI' made writing resource templates for them very difficult. Requirements for new plug-in metadata in Photoshop 3.0 introduced further complexities. The more general and flexible 'PiPL' mechanism was designed to address these issues.
'PiMI' based plug-ins are still fully supported. This is accomplished by converting the 'PiMI' into a 'PiPL' when the plug-in is first scanned. Since 'PiPL's are cached in Photoshop's preferences file, this conversion only happens once.
All plug-in file types are searched for 'PiPL' resources. Historically each type of plug-in had its own file type, as follows:

Plug-in type	Macintosh file type	Windows extension
General (any type of plug-in)	8BPI	.8bp
Acquire modules	8BAM	.8ba
Export modules	8BEM	.8be
Filter plug-ins	8BFM	.8bf
File Format plug-ins	8BIF	.8bi
Accelerator Extensions	8BXM	.8bx
Parser plug-ins	8BYM	.8by

Filenames and extensions are case insensitive.
Only plug-ins of the correct type are searched for 'PiMI' resources of a given type. (This is due to the pairing up of 'PiMI' resources with an appropriate type of code resource.) File types are only a matter of convention for 'PiPL' based plug-ins. All the above file types are searched for 'PiPL' resources and for those that are found, the information contained therein is used to determine the type of plug-in, code location, etc.
If no 'PiPL' resources are found in a plug-in file, the 'PiMI' search algorithm is used as documented in the following section. This allows one to place both 'PiPL' and 'PiMI' resources in a plug-in. 3.0 or later compatible hosts will use the 'PiPL' while 2.5.1 compatible hosts will use the 'PiMI'.

Structure
The Plug-in property list has a version number and a count followed by a sequence of arbitrary length byte containers called properties. A "C " struct definition for the plug-in property list is as follows:

```
        typedef struct PIPropertyList
          {
```

```
        int32 version;
        int32 count;
        PIProperty properties[1];
    } PIPropertyList;
```

- **version**
  This denotes the version of this specification the 'PiPL' is formatted to. The current version is 0.

- **count**
  This field holds the number of properties contained in the 'PiPL'. 0 is a valid value denoting a 'PiPL' with no properties.

- **properties**
  A variable length array of variable length property data structures. Holds the actual contents of the 'PiPL'.

Each property has a vendor code, a key, an ID, a length, and property data the size indicated by the length. The "C" struct definition for the plug-in properties are as follows:

```
   typedef struct PIProperty
    {
        OSType vendorID;
        OSType propertyKey;
        int32  propertyID;
        int32  propertyLength;
        char   propertyData [1];
        /* Implicitly aligned to multiple of 4 bytes. */
    } PIProperty;
```

The fields are defined as follows:

- **vendorID**
  This field identifies the vendor defining this property type. This allows other vendors to define their own properties in a way that does not conflict with either Adobe or other vendors. It is recommended that a registered application creator code be used for the vendorID to ensure uniqueness. All Photoshop properties described in this document use the vendorID '8BIM'.
- **propertyKey**
  This field specifies the type of this property. Property types used by Photoshop are documented below. (Think of a property type as similar to a resource type.)
- **propertyID**
  In theory this can be used to store more than one property of a given type (rather like a resource ID). In practice, this field is always zero. It should be thought of as reserved for future use.

- **propertyLength**
  This field contains the length of the **propertyData** field. It does not include any padding bytes after propertyData to achieve four byte alignment. This field may be zero.
- **propertyData**
  A variable length field that contains the bytes which are the contents of this property. Any values may be contained.

Padding
Each property must be padded such that the next property begins on a four byte boundary.

Notes
Specific properties can be extended in an upward compatible fashion by adding extra data at their end. The length field will allow an application to determine how much data is present, Optional properties can be omitted without concern. (As opposed to a fixed length structure where omitted fields must be given a default value.) The 'PiPL' format is fairly portable in that everything is four byte aligned. All OSType and int32 fields are represented in native byte order for a given platform so the bytes of "the same" 'PiPL' will differ between a big-endian machine (e.g. the Macintosh) and a little-endian machine (e.g. an Intel x86 based Windows machine). Although if one examines the bytes of the PiPL section of an x86 resource binary, they will be backward compared to the Mac, the user can generally not concern themselves with the difference. If they use the pre-defined PI-types, they will be interpreted and stored correctly as in the following example (see PIKindProperty). If, however, an OSType has not been defined and they wish to enter it as a 4-char series, then (since it is not interpreted as a long) they would have to supply the chars in reverse order (see "MIB8").

```
"MIB8",
PIKindProperty,
0L,
4L,
"MFB8",
```

The Macintosh plug-in kit includes a resource template for the 'PiPL' type and the Windows version of the kit includes a "PiPL Parser" application (CnvtPiPL.exe) to transform Mac ".r" files into Windows ".rc" files. If you are developing for the Macintosh platform, you can automatically convert your Macintosh PiPL resources into MSWindows' custom PiPL format by using CnvtPiPL.exe. This enables you to keep just one copy of PiPL and saves you the headache of converting PiPLs by hand and also eliminates the errors caused in the process. In order to use CnvtPiPL.exe, you need to pre-process your *.r file using the standard C pre-processor and pipe the output to CnvtPiPL. The sample makefiles illustrate the process. Even if you are not developing for the Macintosh, you are strongly encouraged to use the resource template (i.e. please refer to any of the *.r files under the "RIncludes" sub-directory) to create the PiPLs, as it is more intuitive to create the PiPLs that way, and then use CnvtPiPL.exe to convert them. CnvtPiPL.exe takes care of all byte alignment and byte-ordering issues for you automatically. If you are going to use the PiPL resource template to create your PiPLs, you can safely ignore the techno-babble about the Windows' PiPL resource format in the following sections.

It is intended for 'PiPL's to collect all plug-in metadata in a single place. This includes the name and category of the plug-in as well as all other information. (In the 'PiMI' world, the name and category were stored as resource names on the plug-in code resource and 'PiMI' respectively.) Vendors are encouraged to define new properties for extensions to plug-in metadata rather than introducing new resource types.

Types

int16, int32:
These are 16 and 32 bit integers respectively. They are stored within the 'PiPL' in native byte order.

OSType:
Same representation as an *int32* but typically denotes a Macintosh style 4 character code like 'PiPL'.

TypeCreatorPair:
A structure of two OSTypes denoting a file type and creator code. The type code is the first field of the structure and the creator code is second.

FlagSet:
This is an array of boolean values where the first boolean is contained in the high order bit of the first byte. The eighth entry would be in the high-order bit of the second byte, etc.

PString:
A Pascal style string where the first byte gives the length of the string and the content bytes follow.

Structures:
Structures are typically represented the same way they would be in memory on the target platform. Native padding and alignment constraints are observed.

Arrays:
Arrays are represented as a contiguous set of entries in the 'PiPL' typically with native padding and alignment constraints observed. The length of the array is usually determined by the property length for arrays of fixed length structures or types.

General properties

Plug-in Kind ***OSType***

```
#define PIKindProperty 0x6b696e64L /* 'kind' */
```

This property encodes the type or kind of a plug-in. Valid values are:

Filter	'8BFM'
File Parser	'8BYM'
File Format	'8BIF'
Accelerator Extension	'8BXM'
Acquire Module	'8BAM'
Export Module	'8BEM'

Version of kind specific API ***int32***

```
#define PIVersionProperty     0x76657273L /* 'vers' */
```

This property encodes a major and minor version number indicating which revision of the plug-in interface this plug-in was written for. The major version number indicates incompatible changes while the minor version number indicates incremental enhancements. The major version number is encoded in the most significant 16 bits of the 32 bit version number, the minor version number is encoded in the least significant 16 bits.

There are separate version numbers for each kind of plug-in. The current version for a given kind of plug-in is defined by a preprocessor macro in the header file defining the interface for that plug-in type.

Plug-in load order priority ***int16***

```
#define PIPriorityProperty     0x70727479L /* 'prty' */
```

This property determines the order in which this plug-in will be loaded. This is typically only important for acceleration extensions. It can however be used to control the order in which items with the same name show up in menus. Lower numbers (including negative ones as the field is signed) load first.

Supported image modes ***FlagSet***

```
#define PIImageModesProperty    0x6d6f6465L /* 'mode' */
```

This is a set of flags that determines which image modes the plug-in supports.

Required Host ***OSType***

```
#define PIRequiredHostProperty  0x686f7374L /* 'host' */
```

This property should be used if a plug-in relies on features of a specific host. It is typically filled in with the applications creator code. (E.g. '8BIM' for Adobe Photoshop.)

Plug-in category ***PString***

```
#define PICategoryProperty   0x63617467L /* 'catg' */
```

Plug-in name ***PString***

```
#define PINameProperty            0x6e616d65L /* 'name' */
```

Code Descriptor Properties

Code descriptors tell Photoshop the type and location of a plug-in's code. More than one code descriptor may be included to build a "fat" plug-in which will run on different types of machines. Photoshop will select the best performing option. Photoshop makes sure that the callback structure is filled in with appropriate functions for the type of code that is loaded. So for PowerPC code, native function pointers will be provided and routine descriptor operations are not required either in calling the plug-in or for the plug-in to invoke Photoshop callback functions.

68k code descriptor ***PI68KCodeDesc***

```
PI68KCodeProperty                0x6d36386bL /* 'm68k' */
```

This descriptor indicates a 68K code resource. The type for this property is as follows:

```
typedef struct PI68KCodeDesc
  {
  OSType resourceType;
  int16 resourceID;
  } PI68KCodeDesc;
```

Any resource type may be used, but conventions for various types of plug-ins are as follows:

Filter	'8BFM'
File Parser	'8BYM'
File Format	'8BIF'
Accelerator Extension	'8BXM'
Acquire Module	'8BAM'
Export Module	'8BEM'

(This convention comes from Photoshop 2.5.1 where these types were required. When building a Plug-in that is backwards compatible with 2.5.1 hosts, these resource types must be used.)

68k FPU code descriptor ***PI68KCodeDesc***

```
PI68KFPUCodeProperty    0x36386670L     /* '68fp' */
```

This descriptor is just like a PI68KCodeDesc except it will only be used on Macintosh machines that are equipped with FPU hardware. This allows vendors to easily ship plug-ins that take advantage of FPU hardware but still run on non-FPU Macs.

PowerPC code fragment descriptor ***PICFMCodeDesc***

```
  PIPowerPCCodeProperty  0x70777063L /* 'pwpc' */
```

This descriptor indicates a PowerPC code fragment in the data fork of the plug-in file. The type for this property is as follows:

```
typedef struct PICFMCodeDesc
{
  long fContainerOffset;
  long fContainerLength;
  char fEntryName[1];
} PICFMCodeDesc;
```

With the fields documented as follows:

- **fContainerOffset**
  Contains the offset within the data fork for the start of this plug-in's code fragment. This allows more than one code fragment based plug-in per file.
- **fContainerLength**
  Holds the length of this plug-ins code fragment. If the fragment extends to the end of the file (e.g. it is the only fragment in the file), the container length may be 0.
- **fEntryName**
  The entrypoint name is represented as a Pascal string and is used to lookup the address of the function to call within the fragment. If the entrypoint name is a zero length string, the default entrypoint for the code fragment will be used. The entrypoint name allows a single code fragment to contain more than one plug-in. (Note: in order for the Code Fragment Manager to find an entrypoint by name, that name must be an exported symbol of the code fragment.) Entrypoint names allow more than one plug-in to be exported from a single code fragment.

Windows 32-bit DLL code descriptor ***PIWin32X86CodeDesc***

```
  #define PIWin32X86CodeProperty 0x77783836L /* 'wx86' */

typedef struct PIWin32X86CodeDesc
{
  "MIB8",
  PIWin32X86CodeProperty,
  0L,
  12L,
  "ENTRYPOINT1\0",  ( if pad needed, "ENTRYPNT1\0\0\0" )
} PIWin32X86CodeDesc;
```

This code descriptor is used for 32 bit Windows DLL's. The entrypoint name is used to lookup the function which is called to invoke the plug-in. The entrypoint name is represented as a NUL terminated string. The string may need to be padded with additional NULs to satisfy the 4 byte alignment requirement.

Note for Windows' Developers:
CnvtPiPL .exe does not recognize any Code Descriptor Property other than "CodeWin32X86".

Filter specific properties

Layer case information ***Array***

```
#define PIFilterCaseInfoProperty 0x66696369L /* 'fici' */
```

The key feature of Photoshop 3.0 is support for dynamically composited layers of image data. A layer consists of color and transparency information for each pixel it contains. Previous versions of Photoshop did not have a transparency component. Transparency introduces a greater richness as well as a number of interesting problems. First off, completely transparent pixels have an undefined color. Second, filters will likely affect transparency data as well as color data. This is especially true for filters which introduce spatial distortions.

Photoshop 3.0 offers a fair bit of flexibility in how transparency data is presented to filters. The filter case info property controls the filtering process and presentation of data to the plug-in. This property provides information to Photoshop about what image data cases the plug-in supports. Photoshop then compares the current filtering situation to the supported cases and chooses the best fitting case. The image data is then presented in that case. If none of the supported cases are usable, the filter will be disabled.

So what are these "cases"? There are seven of them. The property is an array of seven four byte entries, one for each case. The cases are as follows:

```
#define filterCaseFlatImageNoSelection 1
```

This is a background layer or a flat image. There is no transparency data. Nor is there a selection.

```
#define filterCaseFlatImageWithSelection 2
```

No transparency data, but a selection may be present. The selection will be presented as mask data.

```
#define filterCaseFloatingSelection          3
```

Image data with an accompanying mask.

```
#define filterCaseEditableTransparencyNoSelection        4
```

A layer with transparency editting enabled and no selection.

```
#define filterCaseEditableTransparencyWithSelection      5
```

A layer with transparency editting enabled and a selection.

```
#define filterCaseProtectedTransparencyNoSelection       6
```

A layer with transparency editting disabled and no selection.

```
#define filterCaseProtectedTransparencyWithSelection     7
```

A layer with transparency editting disabled and a selection.

Photoshop's fall through algorithm is as follows:
If the editable transparency cases are unsupported, then Photoshop will try the corresponding protected transparency cases. This is important because this governs whether the filter will be expected to filter the transparency data as well as the color data.
If the protected transparency case without a selection is disabled, Photoshop will fall through from there to treating the layer data as a floating selection. As such, the transparency data will be presented via the mask portion of the interface rather than with the input data.

Each four byte entry looks like so:

```
struct caseInfo {
  unsigned8 inputHandling;
  unsigned8 outputHandling;
  unsigned8 flags;
  unsigned8 reserved;
};
```

The inputHandling and outputHandling fields specify pre-processing and post-processing of the image data respectively. Common values for both fields are as follows:

```
#define filterDataHandlingCantFilter        0
```

This case is not supported.

```
#define filterDataHandlingNone     1
```

Do nothing to the image data.

The next three cases are matting cases, which are useful when performing spatial distortions and blurs. You can matte the data, process it, and then dematte to remove the added color. For these cases, the matting is defined as follows:

```
mattedValue =   ((unmattedValue * transparency) + 128) / 255 +
        ((matConstant * (255 - transparency)) + 128) / 255
```

Dematting is defined as follows:

```
unmattedValue =     ((mattedValue - matConstant) ./ transparency) +
        matConstant
```

with the ./ operator defined to be a suitable 8 bit fixed-point divide and the result value being pinned to the range of 0 to 255.

```
#define filterDataHandlingBlackMat         2
```

For the input case, matte the image data with black (0) values based on the transparency. For output, dematte the image data using black (0) values.

```
#define filterDataHandlingGrayMat          3
```

Matte the image data with gray (128) values based on the transparency on input. Dematte the image data using gray (128) values on output.

```
#define filterDataHandlingWhiteMat         4
```

Matte the image data with white (255) values based on the transparency on input. Dematte the image data using white (255) values on output.

The following modes are only useful for input:

```
#define filterDataHandlingDefringe         5
```

Defringe transparent areas filling with the nearest defined pixels using taxicab distance. Note that this only applies to fully transparent pixels.

```
#define filterDataHandlingBlackZap         6
```

Set color component of totally transparent pixels to black (0).

```
#define filterDataHandlingGrayZap          7
```

Set color component of totally transparent pixels to gray (128).

```
#define filterDataHandlingWhiteZap         8
```

Set color component of totally transparent pixels to white (255).

```
#define filterDataHandlingBackgroundZap         10
```

Set color component of totally transparent pixels to the current background color.

```
#define filterDataHandlingForegroundZap         11
```

Set color component of totally transparent pixels to the current foreground color.

The following mode is only useful for output:

```
#define filterDataHandlingFillMask        9
```

This mode results in the transparency mask automatically being filled with full opacity in the area affected by the filter. This is only valid for the editable transparency cases. This option is provided to make it easy to write things like Photoshop's Clouds plug-in which wants to fill an area with a value.

The flags field holds the following bits. (Note: This field is not a FlagSet. The first bit (PIFilterDontCopyToDestinationBit) is in the least-significant bit of the flag byte.)

```
#define PIFilterDontCopyToDestinationBit        0
```

Normally Photoshop copies the source data to the destination before filtering. This gives a good default value for any pixels the filter does not write too, but degrades performance for filters which write all the output pixels. Setting this bit inhibits the copying behavior.

```
#define PIFilterWorksWithBlankDataBit1
```

This flag determines whether the filter will work on "blank" areas. That is, areas that are completely transparent. If not, an error message will be given when the filter is invoked on a blank area. This is only valid for the editable transparency case because that is the only case where we could create opacity -- in the protected transparency case, we would be left with what we started with: completely blank data.

```
#define PIFilterFiltersLayerMaskBit        2
```

In cases where transparency is editable, this flag determines if Layer Masks are filtered. (See the "Add Layer Mask" item in the Layers palette menu to create a layer mask.) Setting this bit adds the layer mask to the set of target channels if: transparency for the layer is editable (i.e., this is one of the editable transparency cases), the bit is set, and the layer mask is specified as being positioned relative to the layer rather than the image in Layer Mask Options. This is the same logic Photoshop uses for built-in filters like blur. The distinction based on position is made with the assumption that layer relative masks will need to be distorted along with the layer while image relative masks are independent of the layer.

Format specific properties

Default file creation information ***TypeCreatorPair***

```
#define PIFmtFileTypeProperty 0x666d5443 /* 'fmTC' */
```

Determines the default type and creator code used for files newly created with this format plug-in. On the Windows platform, we don't store TypeCreator information (except internally), the PIFmtFileTypeProperty is not required, they are simply interpreted as of type 'BINA' and creator 'mdos'. All the info regarding what files can be read/written is obtained from the PIReadExtProperty or the PIFilteredExtProperty. PiMI extensions are converted to PIReadExtProperty's so use of PIFilteredExtProperty requires additional coding if the developer is porting a 16-bit plugin to 32-bit.

Readable types ***TypeCreatorPair[ ]***

```
#define PIReadTypesProperty 0x52645479 /* 'RdTy' */
```

This property contains a list of type and creator pairs which the format plug-in can read.

Filtered types ***TypeCreatorPair[ ]***

```
#define PIFilteredTypesProperty 0x66667454 /* 'fftT' */
```

This property contains a list of type and creator pairs for which the file format plug-in should be called to determine if the file can be read. See documentation for formatSelectorFilterFile plug-in selector.

Readable extensions ***OSType[ ]***

```
#define PIReadExtProperty 0x52644578 /* 'RdEx' */
```

This property contains a list of extensions which the format plug-in can read. The extension is stored in the first three characters of the OSType. The fourth character must be a space.
(The extension ".abc" would be encoded as "abc ". It is 0x61626320 on a big-endian machine. It is byte-reversed on a little-endian machine 0x20636261, but you can still encode it as "abc ".)

Filtered extensions ***OSType[ ]***

```
#define PIFilteredExtProperty 0x66667445 /* 'fftE' */
```

This property contains a list of extensions for which the file format plug-in should be called to determine if the file can be read. See documentation for formatSelectorFilterFile plug-in selector.
(The extension ".abc" would be encoded as "abc ". It is 0x61626320 on a big-endian machine. It is byte-reversed on a little-endian machine 0x20636261, but you can still encode it as "abc ".)

Format flags ***FlagSet***

```
#define PIFmtFlagsProperty 0x666d7466 /* 'fmtf' */
```

This property contains a set of flags which control the operation of file format plug-ins. The default value for any flag is false.

```
		#define PIFmtReadsAllTypesFlag		0
```

obsolete.

```
		#define PIFmtSavesImageResourcesFlag	1
```

Along with the pixel information for a file, Photoshop stores various so-called image resources: printing information, pen tool paths, etc.. Collectively, these are known as image resources. The plug-in format has the option of taking responsibility for these resources by reading and writing a block of data containing the image resources. If this flag is false, Photoshop will add the image resources to the file's resource fork but this will not be portable to other platforms.

```
		#define PIFmtCanReadFlag	2
```

This flag should be set to true if the file format can read files.

```
		#define PIFmtCanWriteFlag	3
```

This flag should be set to true if the file format can write files.

```
		#define PIFmtCanWriteIfReadFlag		4
```

Flag indicating whether we can write using this plug-in if we read the file using this plug-in. For example, the plug-in to support Adobe Premiere's Filmstrip format has the can write flag set to false because it cannot in general be used to save files. It has this flag set to true, however, because we can save out filmstrips that we read in using the plug-in.

Maximum supported size ***Point***

```
		#define PIFmtMaxSizeProperty	0x6d78737a /* 'mxsz' */
```

The maximum number of rows and columns that can be in an image saved in this format. Photoshop will use this field to screen out ineligible formats.

Maximum channels ***int16[ ]***

```
		#define PIFmtMaxChannelsProperty 0x6d786368 /* 'mxch' */
```

An array of one byte counts of the maximum number of channels which can/will be saved for a given image mode. This array is indexed by the plug-in mode constants. For example, if a format supports a single alpha channel in RGB mode, it should set maxChannelsÊ[plugInModeRGBColor] to 4. A plug-in may still be asked to save more channels than it reports it can support. This field exists primarily so that we can warn the user that alpha channels will be discarded.

Parser specific properties

Parsable types ***TypeCreatorPair[ ]***

```
#define PIParsableTypesProperty 0x70735459L /* 'psTY' */
```

This property contains a list of type and creator pairs for files which the parser plug-in can parse.

Filtered parsable types ***TypeCreatorPair[ ]***

```
#define PIFilteredParsableTypesProperty 0x70735479L /* 'psTy' */
```

This property contains a list of type and creator pairs for files which the parser plug-in may be able to parse. The plug-in will be called to make the determination. See the documentation for the parserSelectorCanRead selector.

Parsable extensions ***OSType[ ]***

```
#define PIParsableExtProperty 0x70734558L /* 'psEX' */
```

This property contains a list of extensions for files which the parser plug-in can parse.

(The extension ".abc" would be encoded as "abc ". It is 0x61626320 on a big-endian machine. It is byte-reversed on a little-endian machine 0x20636261, but you can still encode it as "abc ".)

Filtered parsable extensions*OSType[ ]*

```
#define PIFilteredParsableExtProperty 0x70734578L /* 'psEx' */
```

This property contains a list of extensions for files which the parser plug-in may be able to parse. The plug-in will be called to make the determination. See the documentation for the parserSelectorCanRead selector.

(The extension ".abc" would be encoded as "abc ". It is 0x61626320 on a big-endian machine. It is byte-reversed on a little-endian machine 0x20636261, but you can still encode it as "abc ".)

Parsable clipboard types ***OSType[ ]***

```
#define PIParsableClipTypesProperty 0x70734342L /* 'psCB'*/
```

This Macintosh specific property contains a list of clipboard type codes which can be parsed by this plug-in.
'PiMI's

'PiMI' resources are superceded by the previously described 'PiPL' resources, however for compatibility with existing plug-ins, Photoshop 3.0 will still recognize and function with plug-ins containing only 'PiMI's.

The 'PiMI' resource consists of two pieces: general information applicable to all (or most) plug-in types and type specific info. The general information precedes the type specific information. Since the information proceeds serially, however, all fields must be filled in through and including the last field supplied. Generally, a plug-in should either just include the version number information or it should include all of the information documented here.
A "C " struct definition for the 'PiMI' resource is as follows:

```
typedef struct PlugInInfo
{
  short       version;
  short       subVersion
  short       priority;
  short       generalInfoSize;
  short       typeInfoSize;
  short       supportsMode;
  OSType      requireHost;

} PlugInInfo;
```

- **version**
  The major version number for the interface used by the plug-in. This field is required.
- **subVersion**
  The minor version number for the interface used by the plug-in. This field is required.
- **priority**
  The priority which should be associated with this plug-in when it loads. Currently, this is only used for extension modules.
- **generalInfoSize**
  The size of the general plug-in information in this resource.
- **typeInfoSize**
  The size of the type-specific plug-in information in this resource.
- **supportsMode**
  A bitmap describing the image modes supported by the plug-in. This field applies to filter, export, and file format plug-ins. If it is not present, Photoshop will assume that the plug-in supports all image modes. This field is one of the ways Photoshop decides whether to dim plug-ins in menus. Since not all hosts may respect this field, the plug-in should still check that it can handle the image mode it has been requested to process. The bits in the bitmap correspond to the **plugInMode** constants in **PIGeneral.h** (i.e. bit 0 corresponds to bitmaps, bit 1 to grayscale, etc.).
- **requiredHost**
  If the plug-in requires a particular host proc (see below), it should specify the signature for that host proc here. If it does not require a particular host proc, it should fill this field with spaces. Photoshop will decline to load plug-ins which require host procs other than Photoshop's **'8BIM'** proc. Plug-in developers should be aware that again one cannot count on host developers to check this field.

The type specific info is documented in the documentation on the various types of plug-ins.

Note that it is possible to have multiple plug-in modules in a single file, as long as the resource numbers do not conflict (if the modules are of different types, the file type should be set to '8BPI', which is always searched as a special case). In most cases it is not a good idea to place multiple modules in a single file, since it reduces the user's control of which modules are installed. However, there are cases when this should be done. One example is matched acquisition/export, though these frequently correspond to the new file format modules. In such cases, only one of the modules should display an about box, describing both modules. Another example is a set of closely related filters, since the decrease in user control may be offset by an improvement in the ease with which users can do maintenance on their plug-in folders.

Execution

Macintosh

When the user takes an action that causes a plug-in module to be called, Adobe Photoshop opens the resource fork of the file the module resides in, loads the resource into memory, locks it, and calls the routine starting at the first byte of the resource. The Macintosh prototype is:

- **pascal void PlugIn (short selector, Ptr stuff, long *data, short *result);**

Windows

When the user takes an action that causes a plug-in module to be called, Adobe Photoshop does a LoadLibrary call to load the module into memory. For each PiPL resource found in the file, Photoshop calls GetProcAddress (routineName) where "routineName" is the name found under "CodeWin32X86" property to get the routine's address. If the file contains only PiMI resources and no PiPLs, Photoshop does a GetProcAddress for each PiMI resource found in the file looking for the entry point ENTRYPOINT% where % is the integer nameID of the PiMI resource to get the routine's address. Once Photoshop obtains the routine's address it calls the routine following the calling conventions:

- **void Plugin(short selector, void * stuff, void *data, short *result);**

The parameters are to be interpreted as follows:

- **selector**
  This is an integer operation selector code. Selector 0 always means display an about box. The meaning of other values depends on the type of plug-in.

- **stuff**
  This is a pointer to a parameter block. The exact nature of the parameter block depends on the type of plug-in. In the case of the about box selector, this pointer leads to a record containing a single platform specific 32-bit value. In the case of the Macintosh, this field contains no useful data.

- **data**
  This is a pointer to a long integer (32-bit value) which Photoshop will maintain for the plug-in across invocations. One standard use for this field is to store a pointer or handle to a block of memory used to store the plug-in's "global" data. It will be zero the first time the plug-in is called.

- **result**
  This is a pointer to the result code to be returned by the plug-in. A value of zero means that no error has occurred. Any positive value means that execution of the plug-in should stop, but the plug-in has already displayed any appropriate error message. (If the user cancels the operation in any way, the plug-in should return a positive value and not report an error.) A negative value also means that execution of the plug-in stop, but that the host should display its standard error dialog describing the error. Each plug-in type has a one plug-in specific error code (see the header files for details) which can be returned here. Standard Maci OS error codes such as memFullErr (-108) can also be used.

Callback Routines

A number of fields in the various plug-in "**stuff**" structures are callbacks to the host program to provide specific services. A number of these routines are common to multiple plug-in types and are documented here. Those specific to a single plug-in type are documented in that type's documentation. Some of these routines are arranged in suites accessed via a pointer to a table of function pointers. Some of these routines are also new in Photoshop 3.0 and may not be provided by other hosts including earlier versions of Photoshop. If a host does not provide a particular routine or suite, the relevant pointer will be null. Photoshop 3.0 has added an error code to indicate that the host does not supply necessary functionality:

```
#defineerrPlugInHostInsufficient -30900
```

All of the routines use Pascal calling conventions. A complete list can be found in **PIGeneral.h.**
The two routines guaranteed to be present if defined in the plug-in interface record provide checking for command-period (or other requests to abort) and access to a host progress indicator:

TestAbort( )

- **pascal Boolean TestAbort ( );**
  The plug-in should call this function several times a second during long operations to allow the user to abort the operation. If the function returns **TRUE**, the operations should be aborted. As a side effect, this changes the cursor to a watch and moves the watch hands periodically.

UpdateProgress( )

- **pascal void UpdateProgress (long done, long total);**
  The plug-in may call this two-argument procedure periodically to update a progress indicator. The first parameter is the number of operations completed; the second is the total number of operations. This procedure should only be called during the actual main operation of the plug-in, not during long operations during the preliminary user interface.
  Photoshop automatically suppresses display of the progress bar during short operations.

Photoshop 3.0 provides a routine to allow some plug-in types to pass events to Photoshop for processing. For example, when a plug-in receives a deactivate event for one of Photoshop's windows, it is considered polite if the plug-in passes this event on to Photoshop. This routine can also be used to allow Photoshop to do updates of its own windows by passing relevant update and null events to Photoshop. The calling sequence for this routine is:

ProcessEvent( )

- **pascal void ProcessEvent (EventRecord *event);**
  The parameter is actually a generic pointer and the data pointed to will depend on the platform the plug-in is running on.

A general host callback routine may or may not be present. It's functionality and calling sequence is host specific and is defined by the hostType field. The general functionality and calling sequence is identified by the signature supplied with the procedure pointer. Host proc's are used to support operations which are special to some host and would not apply to most other hosts. In the case of Photoshop's callback, for example, this is a place where some callback operations which may well not be supported in the long term but which are critical to getting certain features in Photoshop working as plug-ins are provided. *This callback is not generally useful on the Windows platform.*

The next general callback routine is used to display pixels in various image modes. It takes a structure describing a block of pixels to display:

DisplayPixels( )

- **pascal OSErr DisplayPixels (const PSPixelMap *source, const VRect *srcRect, int32 dstRow, int32 dstCol, unsigned32 platformContext);**

The parameters have the following interpretations:

- **source**
  The **PSPixelMap** containing the pixels to be displayed.

```
typedef struct PSPixelMap
  {
      int32 version;
      VRect bounds;
      int32 imageMode;
      int32 rowBytes;
      int32 colBytes;
      int32 planeBytes;
      void *baseAddr;
  /* Fields new in version 1. */
      PSPixelMask *mat;
      PSPixelMask *masks;
      int32 maskPhaseRow;
      int32 maskPhaseCol;
} PSPixelMap;
```

  The fields in this structure are as follows:

  - **version**
    The version number for this structure. The current version number is version 1. Future versions of Photoshop may support additional parameters and will support higher version numbers for PSPixelMap's.

- **bounds**
  The bounds for the pixel map.
- **imageMode**
  The mode for the image data. The supported modes are grayscale, RGB, CMYK, and Lab. Additionally, if the mode of the document being processed is DuotoneMode or IndexedColorMode, you can pas plugInModeDuotone or plugInModeIndexedColor.
- **rowBytes**
  The offset from one row to the next of pixels.
- **colBytes**
  The offset from one column to the next of pixels.
- **planeBytes**
  The offset from one plane of pixels to the next. In RGB, the planes are ordered red, green, blue; in CMYK, the planes are ordered cyan, magenta, yellow, black; in Lab, the planes are ordered L, a, b.
- **baseAddr**
  The address of the byte value for the first plane of the top left pixel.
- **mat**
  For all modes except indexed color, you can specify a mask to be used for matting correction. For example, if you have white matted data to display, you can specify a mask in this field which will be used to remove the white fringe. This field points to a PSPixelMask structure (see below) with a maskDescription indicating what type of matting needs to be compensated for. If this field is NULL, Photoshop performs no matting compensation. If the masks are chained, only the first mask in the chain is used.
- **masks**
  This points to a chain of PSPixelMasks which are multiplied together (with the possibility of inversion) to establish which areas of the image are transparent and should have the checkerboard displayed. kSimplePSMask, kBlackMatPSMask, kWhiteMatPSMask, and kGrayMatPSMask all operate such that 255 = opaque and 0 = transparent. kInvertPSMask has 255 = transparent and 0 = opaque.
  The PSPixelMasks structure is defined as follows:

```
typedef struct PSPixelMask
  {
  struct PSPixelMask *next;
  void *maskData;
  int32 rowBytes;
  int32 colBytes;
  int32 maskDescription;
  }
PSPixelMask;
```

        - **next**
          A pointer to the next mask in the chain

        - **maskData**
          A pointer to the mask data.

        - **rowBytes**
        - **colBytes**
          The row and column steps for the mask.

        - **maskDescription**
          The mask description value, which is one of the following:

```
#define kSimplePSMask   0
#define kBlackMatPSMask 1
#define kGrayMatPSMask  2
#define kWhiteMatPSMask 3
#define kInvertPSMask   4
```

    - **maskPhaseRow**
    - **maskPhaseCol**
      maskPhaseRow and maskPhaseCol give the phase of the checkerboard with respect to the top left corner of the **PSPixelMap**.

- **srcRect**
  The rectangle within that **PSPixelMap** to be displayed.

- **dstRow**
- **dstCol**
  The coordinates of the top left destination pixel in the current port (i.e., the destination pixel which will correspond to the top left pixel in srcRect). The display routines does not scale the pixels, so specifying the top left corner is sufficient to specify the destination.

- **platformContext**
  This parameter is not used on the Macintosh since the routine simply assumes that the target is the current port. On Windows, platformContext should be the target hDC, cast to an unsigned32. Under other platforms, this may specify a drawing context.

The routine will do the appropriate color space conversion and copybits the results to the screen with dithering. It will leave the original data intact. If it is successful, it will return **noErr**. Non-success is generally due to unsupported color modes.

GetPropertyProc( )

The GetProperty callback is available to filter and export modules. It allows these modules to get information about the document currently being processed.

- **pascal OSErr (*GetPropertyProc) (OSType signature, OSType key, int32 index,
int32 * simpleProperty, Handle *complexProperty);**

The signature and key form a pair to identify the property of interest. Photoshop's signature is always '8BIM'. The key values are documented below and in PIProperties.h.
Properties like channel names and path names or data can be indexed. Indices generally start at 0.
Properties can consist either of an integer returned in simpleProperty or a handle returned in complexProperty. The type of property data is documented in PIProperties.h. In the case of a complex (i.e., handle based) property, the plug-in is responsible for disposing of the handle it is passed via the dispose call in the handle suite.
Properties involving strings -- e.g., channel names and path names -- are returned in a handle where the length of the handle determines the size of the string. There is no length byte nor is the string zero terminated.

Property keys

'nuch' (number of channels): This returns the number of channels in the document. This count will include the transparency mask and the layer mask for the target layer if these are present. This property is simple.

'nmch' (channel name): This returns the name of the channel. The channels are indexed from zero and consist of the composite channels, the transpareny mask, the layer mask, and the alpha channels. This property is complex.

'mode' (image mode): This returns the mode of the image using the constants defined in PIGeneral. This property is simple.

'nupa' (number of paths): This property returns the number of paths in the document. This property is simple.

'nmpa' (path name): This property returns the name of the indexed path. The paths are indexed starting with zero. This property is complex.

'path' (path contents): This property returns the contents of the indexed path in the format documented in the path resources documentation. LittleEndian platforms should note that the data is stored in BigEndian form. This property is complex.

'wkpa' (work path index): This property returns the index of the work path or -1 if there is no work path. This property is simple.

'clpa' (clipping path index): This property returns the index of the clipping path or -1 if there is no clipping path. This property is simple.

'tgpa' (target path index): This property returns the index of the target
path or -1 if there is no target path. This property is simple.

This list is complete for Photoshop 3.0.1. Future versions of the program will almost certainly support more properties. We will also probably add a way to write to properties from plug-ins in some future version. There is no way to do so at this time.

AdvanceStateProc ( )

- **pascal OSErr (*AdvanceStateProc) (void);**

In a plug-in type specific manner, this callback allows the plug-in to drive Photoshop to update the buffers used for communicating between Photoshop and the plug-in without the plug-in actually returning from the selector call. It returns noErr if successful and a non-zero error code if something went wrong.

ColorServicesProc ( )

- **pascal OSErr (*ColorServicesProc) (ColorServicesInfo *info);**

```
typedef struct ColorServicesInfo
{
        int32 infoSize;
        int16 selector;
        int16 sourceSpace;
        int16 resultSpace;
        Boolean resultGamutInfoValid;
        Boolean resultInGamut;
        void *reservedSourceSpaceInfo;
        void *reservedResultSpaceInfo;
        int16 colorComponents[4];
        void *reserved;
        Str255 *pickerPrompt;
}
ColorServicesInfo;
```

The fields of this record are as follows:

- **infoSize**
  This field must be filled in with the size of the ColorServicesInfo record in bytes. The value is used as a version identifier in case this record is expanded in the future. It can be filled in like so:
  ```
  ColorServicesInfo requestInfo;
  requestInfo.infoSize = sizeof(requestInfo);
  ```

- **selector**
  This field selects the operation performed by the ColorServices callback. At present there are two operations available, choosing a color using the Photoshop color picker (actually, using the user's preferred color picker), and converting color values from one color space to another. The selectors for these are respectively:

  ```
  #define plugIncolorServicesChooseColor  0
  ```

```
#define plugIncolorServicesConvertColor 1
```

Available color spaces:

```
#define plugIncolorServicesRGBSpace     0
#define plugIncolorServicesHSBSpace     1
#define plugIncolorServicesCMYKSpace    2
#define plugIncolorServicesLabSpace     3
#define plugIncolorServicesGraySpace    4
#define plugIncolorServicesHSLSpace     5
#define plugIncolorServicesXYZSpace     6
```

- **sourceSpace**
  This field is used for to indicate the color space of the input color contained in colorComponents. For plugIncolorServicesChooseColor the input color is used as an initial value for the picker. For plugIncolorServicesConvertColor the input color will be converted from the color space indicated by sourceSpace to the one indicated by resultSpace.

- **resultSpace**
  This field holds the desired color space of the result color from the ColorServices call. The result will be contained in the colorComponents field when ColorServices returns. For the plugIncolorServicesChooseColor selector, resultSpace can be set to plugIncolorServicesChosenSpace to return the color in whichever color space the user chose the color. In that case, resultSpace will contain the chosen color space on output.

- **resultGamutInfoValid**
  This output only field indicates whether the resultInGamut field has been set. In Photoshop 3.0, this will only be true for colors returned in the plugIncolorServicesCMYKSpace color space.

- **resultInGamut**
  This output only field is a boolean value that indicates whether the returned color is in gamut for the currently selected printing setup. It is only meaningful if the resultGamutInfoValid field is true.

- **colorComponents**
  This array contains the actual color components of the input or output color. They will be included in the components array as they are listed in the color space name. So for plugIncolorServicesRGBSpace colorComponents[0] will contain the red (R) component, colorComponents[1] will contain the green (G) component, colorComponents[2] will contain the blue (B) component. Components not used in the input color space need not be filled in and components not used in the result color space are undefined.

- **pickerPrompt**
  This field contains a pointer to a Pascal string which will be used as a prompt in the Photoshop color picker for the plugIncolorServicesChooseColor call. NULL can be passed to indicate no prompt should be used.

- **reservedSourceSpaceInfo**
- **reservedResultSpaceInfo**
- **reserved**

These three fields are reserved for future expansion and must be set to NULL. A parameter error will be returned if they are not.

Monitor Descriptions

A number of the plug-ins get passed monitor descriptions via the **PlugInMonitor** structure. These descriptions basically detail the information recorded in Photoshop's Monitor Setup dialog and are passed in a structure of the following type:

```
typedef struct PlugInMonitor
{
  Fixed gamma;
  Fixed redX;
  Fixed redY;
  Fixed greenX;
  Fixed greenY;
  Fixed blueX;
  Fixed blueY;
  Fixed whiteX;
  Fixed whiteY;
  Fixed ambient;

} PlugInMonitor;
```

The fields of this record are as follow:

- **gamma**

  This field contains the monitor's gamma value or zero if the whole record is invalid.

- **redX**
- **redY**
- **greenX**
- **greenY**
- **blueX**
- **blueY**

  These fields specify the chromaticity coordinates of the monitor's phosphors.

- **whiteX**
- **whiteY**

  These fields specify the chromaticity coordinates of the monitor's white point.

- **ambient**
  This field specifies the relative amount of ambient light in the room. Zero means a relatively dark room, 0.5 means an average room, and 1.0 means a bright room.

Callback Suites

The rest of the callback routines are organized into "suites", collections of related routines which implement a particular functionality. The suites are described by a pointer to a record containing in order a 2 byte version number for the suite, a 2 byte count of the number of routines in the suite - routines can be added to the suite without incrementing the version number - and a series of **ProcPtr**'s for the routines. Before calling a callback defined in the suite, the plug-in needs to check the following conditions:

- The suite pointer must not be null.
- The suite version number must match the version number the plug-in wishes to use. (We do not expect to be changing version numbers with any degree of frequency.)
- The number of routines defined in the suite must be great enough to include the routine of interest.
- The pointer for the routine of interest must not be null.

If these conditions are not met and the plug-in does not want to work around the non-availability of the callback, the plug-in should put up an error dialog and then return to the host as if the user had canceled.

Buffer Suite

The buffer suite provides an alternative to the memory management functions available in previous versions of Photoshop's plug-in specification by providing a set of routines to request that the host allocate and dispose of memory out of a pool which it manages.

Photoshop 2.5, for example, goes to a fair amount of trouble to balance the need for buffers of various sizes against the space needed for the tiles in its virtual memory system. Growing the space needed for buffers will result in Photoshop shrinking the number of tiles it keeps in memory.

Previous versions of the plug-in specification provide some mechanisms for interacting with this system by letting a plug-in specify a certain amount of memory which the host should reserve for the plug-in. This approach has two problems: (1) the memory is reserved throughout the execution of the plug-in and (2) the plug-in may still run up against limitations imposed by the host - for example, Photoshop 2.5 will, in large memory configurations, allocate most of memory at startup via a **NewPtr** call, and this memory will never be available to the plug-in other than through the buffer suite. On Windows, Photoshop's memory scheme is designed such that it allocates just enough memory to not let Windows' virtual memory manager to kick in. If the plug-in allocates lots of memory using ***GlobalAlloc ( )***, this scheme will be defeated and Photoshop will be double-swapping thereby degrading performance. Using the buffer suite, a plug-in can avoid doing some of the accounting for space to be reserved. This simplifies the prepare phase for acquire, filter, and format plug-ins. Unfortunately, export modules are expected to account for the buffer for the data requested from the host even though the host allocates the buffer. This means that the buffer suite routines do not really provide any help for export modules. But for other plug-ins, buffer allocations can be delayed until they are actually needed.

Buffers are identified by pointers to an opaque type called **BufferID**'s.

Version 1 was purely developmental. The routines in version 2 of the suite are:

AllocateBuffer( )

- **pascal OSErr AllocateBuffer (int32 size, BufferID *buffer);**
  This routine sets buffer to be the ID for a buffer of the requested size and returns noErr if allocation is successful. It returns an error code if allocation is unsuccessful. Note that buffer allocation is more likely to fail during phases where other blocks of memory are locked down for the plug-in's benefit - e.g., during the continue calls to filter and format plug-ins.

LockBuffer( )

- **pascal Ptr LockBuffer (BufferID buffer, Boolean moveHigh);**
  This locks the buffer so that it won't move in memory and returns a pointer to the beginning of the buffer. It will optionally try to move the block to the high end of memory to avoid fragmentation. "moveHigh" parameter has no effect under MS-Windows.

UnlockBuffer( )

- **pascal void UnlockBuffer (BufferID buffer);**
  This is the corresponding routine to unlock a buffer. A buffer can be locked multiple times and only the final balancing unlock call will actually unlock it.

FreeBuffer( )

- **pascal void FreeBuffer (BufferID buffer);**
  This routine releases the storage associated with a buffer. Use of the buffer's ID after calling **FreeBuffer** will probably result in severe crashes.

BufferSpace( )

- **pascal int32 BufferSpace (void);**
  This routine returns the amount of space available for buffers. This space may be fragmented so an attempt to allocate all of the space as a single buffer may fail.

Pseudo-Resource Suite

Macintosh only.
The second suite of callback routine provides support for storing data with and retrieving data from a document. These routines essentially provide pseudo-resources which plug-ins can attach to documents and use to communicate with each other. Each resource is a handle of data and is identified by a 4 character code (ResType) and a one-based index. (**NOTE: Some sort of registry needs to be set up for resource types. No such registry yet exists. For now, please contact AppleLink: ADOBE.WISE or wise@mv.us.adobe.com to discuss registering a type. **)
The first fully functional version of the suite is version 3. The routines in that version are:

CountPIResources( )

- **pascal int16 CountPIResources (ResType ofType);**
  This routine returns a count of the number of resources of a given type.

GetPIResource( )

- **pascal Handle GetPIResource (ResType ofType, int16 index);**
  This routine returns the indicated resource for the current document or NULL if no resource exists with that type and index. The handle returned belongs to the host and should be treated as a read only handle.

DeletePIResource( )

- **pascal void DeletePIResource (ResType ofType, int16 index);**
  This routine deletes the resource that would have been returned by **GetPIResource**. Note that since resources are identified by index rather than ID, this will cause subsequent resources to renumber.

AddPIResource( )

- **pascal OSErr AddPIResource (ResType ofType, Handle data);**
  This routine adds a resource of the given type at the end of the list for that type. The contents of **data** are duplicated so that the plug-in retains control over the original handle. If there is not enough memory or the document already has too many plug-in resources (the limit in Photoshop is 1000), this routine will return **memFullErr**.

Handle Suite

The use of handles in the pseudo-resource suite poses a problem for platforms other than the Macintosh where a direct equivalent may not exist. In those cases, Photoshop chooses a specific model for what it expects of a handle. The following suite of routines is used primarily for cross-platform support purposes since on the Macintosh, handles are handles. The one additional feature gained by using these routines rather than the Macintosh toolbox is that Photoshop will account for these handles in its VM space calculations. Hence, it is important to free any handles allocated using this suite by calling the free routine provided in this suite, etc..
Here are the routines in version 1 of the suite:

NewPIHandle ( )

- **pascal Handle NewPIHandle (int32 size);**
  This routine allocates a handle of the indicated size. It returns NULL if the handle could not be allocated.

DisposePIHandle ( )

- **pascal void DisposePIHandle (Handle h);**
  This routine disposes of the indicated handle.

GetPIHandleSize ( )

- **pascal int32 GetPIHandleSize (Handle h);**
  This routine returns the size of the indicated handle.

SetPIHandleSize ( )

- **pascal OSErr SetPIHandleSize (Handle h, int32 newSize);**
  This routine attempts to resize the indicated handle. It returns noErr if successful and an error code if unsuccessful.

LockPIHandle ( )

- **pascal Ptr LockPIHandle (Handle h, Boolean moveHigh);**
  This routine locks and dereferences the handle. Optionally, the routine will move the handle to the high end of memory before locking it. This routine really only matters for cross platform implementations.

UnlockPIHandle ( )

- **pascal void UnlockPIHandle (Handle h);**
  This routine unlocks the handle. Unlike the routines for buffers, the lock and unlock calls for handles do not nest - a single unlock call unlocks the handle no matter how many times it has been locked. This routine really only matters for cross platform implementations.

RecoverSpaceProc ( )

- **pascal void (*RecoverSpaceProc) (int32 size);**

All handles allocated through the Handle Suite have their space accounted for in Photoshop's estimates of how much image data it can make resident at one time. If you obtain a handle via the handle suite (or some other mechanism in Photoshop) which you are supposed to dispose of using the DisposePIHandle callback but instead dispose of in some other way (e.g., use the handle as the parameter to AddResource and then close the resource file), then you can use this call to tell Photoshop to stop reserving space for the handle.

General Notes

Macintosh

Global Variables

Most Macintosh development systems reference global variables by using negative offsets from register A5. If a plug-in were to try to use global variables in the standard way, its global variable space would overlap Adobe Photoshop's global variable space, usually resulting in a quick and fiery death.
The solution is to write the code in such a way as to not require global variables. In most cases, it is possible to replace global variables with additional procedure parameters. One case where this is impossible is with static data, which must be preserved between calls to the plug-in. Static data can be stored by allocating a handle, storing the static data in memory pointed to by the handle, and storing the handle in the data parameter, which Adobe Photoshop preserves between calls. This is the approach taken in the sample plug-in code with this kit since it allows the code to work equally well under any development environment.

Segmentation

Macintosh 680x0 applications have a special code segment called the jump table. When a routine in one segment calls a routine in another segment, it actually calls a small glue routine in the jump table segment. This glue routine loads the routine's segment into memory if needed, and jumps to its actual location.
The jump table is accessed using positive offsets from register A5. Since Photoshop is already using A5 for its jump table, the plug-in cannot use a jump table in the standard way.
The simplest way to solve this is to link all the plug-in's code into a single segment. This usually requires the setting of optional compilation/link flags under most development environments if the resultant segment exceeds 32k.

About Boxes

All five kinds of plug-in should respond to a selector value of zero, which means display an about box. The plug-in actually has complete freedom to display any kind of about box it wishes, but to fit in smoothly with the Adobe Photoshop interface it should obey the following conventions:

1. It should be centered on the main (menu-bar) screen, with 1/3 of the remaining space above the dialog, and 2/3 below. Be sure to take into account the menu bar height.
2. It should not have an OK button, but should instead respond to a click anywhere in its dialog.
3. It should respond to the return and enter keys.

If you have placed multiple plug-in modules in a single file, only one of them should display an about box, which should describe all of the modules. When Photoshop attempts to bring up the about box for a plug-in, it will make the about box call for all of the plug-ins in the same file.

Configuration

Photoshop plug-ins may assume 128K or larger ROMs, and System 6.0.2 or later. PiPL-only plug-ins (i.e., Photoshop 3.0 or later) may assume System 7. Keep in mind that Photoshop will run, and thus your plug-in may be called, on machines as old as the Mac Plus. Thus, plug-ins should not assume, and should check for if they require: 68020 or 68030 processors, math co-processors, 256K ROMs, and Color or 32-Bit QuickDraw. Photoshop 3.0 requires a 68020 or better, Color QuickDraw, and 32-Bit QuickDraw, so if you restrict your plug-in so that it only runs under Photoshop 3.0, you can assume these features are present.

Windows

Configuration

Photoshop plug-ins may assume Windows 3.1 in standard or enhanced mode and a 80386 processor.

About the Sample Plug-ins

Macintosh Version

The 6 sample plug-ins included with this kit can be built using MPW. They have been tested against MPW 3.3.1.

The kit includes new header files. PIGeneral.h and PITypes.h contain definitions useful across multiple plug-ins. PIAbout.h contains the information for the about box call for all plug-in types. PIAcquire.h, PIExport.h, PIFilter.h, and PIFormat.h are the header files for the respective types of plug-in modules.

Also included are two sets of utilities: DialogUtilities and PIUtilities.

DialogUtilities.c and DialogUtilities.h provide general support for doing things with dialogs including creating movable modal dialogs which make appropriate calls back to the host to update windows and such.

PIUtilities.c and PIUtilities.h contain various routines and macros to make it easier to use the host callbacks. The macros make various assumptions about how global variables are being handled. None of the routines worry about switching A5 worlds since the sample plug-ins do not use A5 worlds. If you do not follow the model for dealing with globals (basically not using them) used in the sample plug-in code, you will probably have to modify these files. Remember, it is VERY bad to call back to the host with the wrong A5 world!

Windows Version

The kit includes two sets of utilities: PIUtilities and Windows Utilities.

PIUtilities.c and PIUtilities.h contain various routines and macros to make it easier to use the host callbacks. The macros make variou assumptions about how global variables are being handled. If you do not follow the model for dealing with globals (basically not using them) used in the sample plug-in code, you will probably have to modify these files.

Winutils.c provides support for some Mac Toolbox functions used in PIUtilities.c, namely memory management functions (e.g NewHandle( ) etc.)

Structure packing for all records (i.e FilterRecord, FormatRecord, AcquireRecord, ExportRecord and AboutRecord) should be the default for the machine (this has changed for 32-bit plugins for speed reasons) however the Info structures (FilterInfo, FormatInfo etc.) must be packed to byte boundaries. This means the PiMi resource should be byte aligned as before. These packing changes are reflected in the appropriate header files using #pragma pack(1) to set byte packing and #pragma pack( ) to restore default packing. These pragmas work only on Microsoft Visual C++ and Windows 32 bit SDK environment tools. If you are using a different compiler, like say Symantec C++ or Borland C++, you have to modify the header files with appropiate pragmas. The Borland #pragmas still appear in the header files as they did in the 16-bit plugin kit, but are untested.

You need a DLLInit ( ) function prototyped as

```
BOOL APIENTRY DLLInit(HANDLE, DWORD,LPVOID);
```

The actual name of this entry point is provided to the linker by the

```
PSDLLENTRY=DLLInit
```

assignment in the sample makefiles.

The way that messages are packed into wParam and lParam have changed for Win32. You will need to insure that your window procedures extract the appropriate information correctly. A new header file "WinUtil.h" defines all the Win32 message crackers for cross-compilation or you may simply change your extractions to the Win32 versions. (See The Win32 Application Programming Interface: An Overview for more information on Win32 message parameter packing.)

Be sure that the definitions for your Windows callback functions (dialog box functions, etc.) conform to the Win32 model. The most common problem is the use of "WORD wParam" for callback functions. The plugin examples use

```
    BOOL WINAPI MyDlgProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
```

which will work correctly for both 16 and 32 bit compilation.

The Windows kit also includes 2 executable utilties, namely, MacToDos.exe and CnvtPiPL.exe.
MacToDos.exe lets you convert Macintosh text files into PC text files. If you happen to pick up your plug-in kit from Adobe server, some source files may be in Macintosh format. You may have to convert them into PC format before compiling them.
CnvtPIPL.exe lets you convert PiPL resource in Macintosh format (ASCII format which conforms to the PiPL resource template) into Windows's PiPL format.
These two utilities are included under the Util sub-directory.

Acquisition Modules

Basics
The code resource and file type for acquisition modules is '8BAM'.

The AcquireRecord Structure
The stuff parameter contains a pointer to a structure of the following type:

```
typedef struct AcquireRecord
{
	int32			serialNumber;
	TestAbortProc	abortProc;
	ProgressProc	progressProc;
	int32			maxData;

	int16			imageMode;
	Point			imageSize;
	int16			depth;
	int16			planes;

	Fixed			imageHRes;
	Fixed			imageVRes;

	LookUpTable		redLUT;
	LookUpTable		greenLUT;
	LookUpTable		blueLUT;

	void *			data;

	Rect			theRect;
	int16			loPlane;
	int16			hiPlane;
	int16			colBytes;
	int32			rowBytes;
	int32			planeBytes;

	Str255			fileName;
	int16			vRefNum;
	Boolean			dirty;

	OSType			hostSig;
	ProcPtr			hostProc;

	int32			hostModes;
```

```
        int16                   planeMap [16];

        Boolean                 canTranspose;
        Boolean                 needTranspose;

        Handle              duotoneInfo;

        int32                   diskSpace;

        SpaceProc           spaceProc;

        PlugInMonitormonitor;

        void *              platformData;

        BufferProcs *  bufferProcs;
        ResourceProcs *   resourceProcs;

        ProcessEventProc processEvent;

        Boolean                 canReadBack;
        Boolean                 wantReadBack;

        Boolean                 acquireAgain;

        Boolean                 canFinalize;

        DisplayPixelsProc displayPixels;

        HandleProcs * handleProcs;

        Boolean             wantFinalize;

        char            reserved1[3];

        ColorServicesProc colorServices;

        AdvanceStateProc advanceState;

        char            reserved [216];

    } AcquireRecord;
```

Record Fields

- **serialNumber**

This field contains Adobe Photoshop's serial number. Plug-in modules can use this value for copy protection, if desired.

- **abortProc**
  This field contains a pointer to the **TestAbort** callback documented in the general documentation.

- **progressProc**
  This field contains a pointer to the UpdateProgress callback documented in the general documentation. This procedure should only be called during the actual main operation of the plug-in, not during long operations during the preliminary user interface. For example, it should not be used during a preview operation that computes a low resolution preview image for cropping. It should be used during the main, high-resolution scan.

- **maxData**
  Photoshop initializes this field to the maximum of number of bytes it can free up. The plug-in may reduce this value during the **acquireSelectorPrepare** routine. The **acquireSelectorContinue** routine should return the image in strips no larger than **maxData**, less the size of any large tables or scratch areas it has allocated unless it uses the buffer or handle suites to allocate the memory.

- **imageMode**
  The **acquireSelectorStart** routine should set this field to inform Photoshop what mode image is being acquired (grayscale, RGB Color, etc.). See the header file for possible values.

- **imageSize**
  The **acquireSelectorStart** routine should set this field to inform Photoshop of the image's width (**imageSize.h**) and height (**imageSize.v**) in pixels.

- **depth**
  The **acquireSelectorStart** routine should set this field to inform Photoshop of the image's resolution in bits per pixel per plane. The only valid settings are 1 for bitmap mode images, and 8 for all other modes except grayscale and RGB which also allow 16.

- **planes**
  The **acquireSelectorStart** routine should set this field to inform Photoshop of the number of channels in the image. For example, if an RGB image without alpha channels is being returned, this field should be set to 3. Because of the implementation of the plane map, acquire modules (and format modules) should never try to work with more than 16 planes at a time. The results would be unpredictable.

- **imageHRes**
- **imageVRes**
  The **acquireSelectorStart** routine should set these fields to inform Photoshop of the image's horizontal and vertical resolution in terms of pixels per inch. This is a fixed point number (16 binary digits). Photoshop initializes these fields to 72 pixels per inch.
  The current version of Photoshop only supports square pixels, so it ignores the **imageVRes** field. Plug-ins should set both fields anyway in case future versions of Photoshop support non-square pixels.

- **redLUT**
- **greenLUT**
- **blueLUT**
  If an indexed color mode image is being returned, the **acquireSelectorStart** routine should return the image's color table in these fields.

- **duotoneInfo**
  If the plug-in is acquiring a duotone mode image, it should allocate a handle and return the duotone information here. The format of the information is the same as that provided by export modules, and should be treated as a black box by plug-ins.
  The plug-in is responsible for freeing the handle in its **acquireSelectorFinish** routine.

- **data**
  The **acquireSelectorContinue** routine should return a pointer to the image's data in this field. After all of the image has been returned, the **acquireSelectorContinue** should set this field to NULL.
  Note that the plug-in is responsible for freeing any memory pointed to by this field. This is a change from previous version's of Photoshop's plug-in interface.

- **theRect**
  The **acquireSelectorContinue** routine should set this field to the area being returned.

- **loPlane**
- **hiPlane**
  The **acquireSelectorContinue** routine should set these fields to the first and last planes being returned. For example, if interleaved RGB data is being returned, they should be set to 0 and 2, respectively.

- **colBytes**
  The **acquireSelectorContinue** routine should set this field to the offset in bytes between columns of returned data. This is usually 1 for non-interleaved data, or (hiPlane - loPlane + 1) for interleaved data.

- **rowBytes**
  The **acquireSelectorContinue** routine should set this field to the offset in bytes between rows of returned data.

- **planeBytes**
  The **acquireSelectorContinue** routine should set this field to the offset in bytes between planes of returned data. This field is ignored if **loPlane** = **hiPlane**. It should be set to 1 for interleaved data.

- **planeMap**
  This is initialized by the host to a linear map (planeMap [i] = i). This is used to map plane (channel) numbers between the plug-in and the host. For example, Photoshop stores RGB images with an alpha channel in the order RGBA, whereas most frame buffers store the data in ARGB order. To return the data in this order, the plug-in should set planeMap [0] to 3, planeMap [1] to 0, planeMap [2] to 1, and planeMap [3] to 2. Note that attempts to index past the end of a planeMap will result in the identity map being used for the indexing.

- **fileName**
  By default, Photoshop opens newly acquired images as "Untitled-..." . File importing acquisition modules should set this field to the file's name in their **acquireSelectorStart** routines, so Photoshop can display the correct window title. Scanning modules should ignore this field.

- **vRefNum**
  If the plug-in sets fileName, it should also set vRefNum to the file's volume reference number. Not applicable on MS-Windows.

- **dirty**
  By default, newly acquired images are marked as dirty, meaning that the user will be prompted to save the unsaved changes when closing the file. File importing acquisition modules should set this field to false to prevent this.

- **hostSig**
  The host program's signature. Photoshop's signature is **'8BIM'**.

- **hostProc**
  If not null, this field contains a pointer to a host-defined callback procedure that can do anything the host wishes. Plug-ins should verify **hostSig** before calling this procedure. This provides a mechanism for hosts to extend the plug-in interface to support application specific features.

- **hostModes**
  This field is used by the host to inform the plug-in which **imageMode** values it supports. If the corresponding bit (LSB = bit 0) is 1, the mode is supported. This field can be used by plug-ins to disable features (such as color scanning) if not supported by the host.

- **canTranspose**
  If the host supports transposing images during or after scanning, it should set this field to true. Photoshop always sets this field to true.

- **needTranspose**
  This field is initialized by the host to false. If the plug-in wishes to have the image transposed, and **canTranspose** is true, it should set this field to true during its **acquireSelectorStart** routine.
  The logical effect is to transpose the image after scanning is complete, although some hosts may find it more efficient to transpose the data during scanning.
  This feature was added to the plug-in specification because versions of Photoshop prior to Photoshop 2.5 had a strong bias toward horizontal strips. Using this routine, a plug-in could acquire an image in vertical strips by passing Photoshop horizontal strips and then having Photoshop transpose the data when it was done.

- **diskSpace**
  This field contains the number of free bytes on the host's scratch disk or disks. If the host does not use a scratch disk, it should set this field to -1.

- **spaceProc**

If not null, this field contains a pointer to a function with the following calling conventions:

- **pascal int32 SpaceProc (void);**

This function examines **imageMode**, **imageSize**, **depth**, and **planes** and returns the number of bytes of scratch disk space required to hold the image. Returns -1 if the values are not valid.

- **monitor**
  This field contains the monitor setup information for the host. See the general documentation for more details.

- **platformData**
  This field contains a pointer to platform specific data. Not used on the Macintosh.

- **bufferProcs**
  This field contains a pointer to the buffer suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **resourceProcs**
  This field contains a pointer to the pseudo-resource suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **processEvent**
  This field contains a pointer to the **ProcessEvent** callback documented in the general documentation. It contains null if the callback is not supported. This function is not useful on Windows.

- **canReadBack**
  If the host supports acquire modules reading back image data for further processing, it should set this field to true. Photoshop always sets this field to true.

- **wantReadBack**
  If the plug-in sets this flag and the host supports image read back for acquire modules, then the host will ignore the contents of the buffer it is passed and will instead fill the buffer with the image data. It will store the data in the format described by **loPlane**, **hiPlane**, **colBytes**, **rowBytes**, **planeBytes**, and **planeMap**. If theRect exceeds the bounds of the image, those portions of the buffer will simply be left untouched.

- **acquireAgain**

If the plug-in wishes to be called again to acquire another image, it should set this flag during the **acquireSelectorFinish** call. Host's that support multiple image acquisition should start the acquisition process again with a call to **acquireSelectorStart** to begin acquiring a new image. Plug-ins which do not want to put up a user interface for each acquisition should put up their interface during the **acquireSelectorPrepare** call. Plug-ins should not count on being called again just because they set this flag - i.e., **acquireSelectorFinish** should still do all of the necessary clean-up. With the addition of the finalize selector, plug-ins can now put up an interface that survives across multiple acquisitions.

- **canFinalize**
  If the host can make the finalize call, it should set this field to true.

- **displayPixels**
  This field contains a pointer to the DisplayPixels callback documented in the general documentation. It contains null if the callback is not supported.

- **handleProcs**
  This field contains a pointer to the handle suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **wantFinalize**
  This flag requests an acquireSelectorFinalize call if the host provides the newer protocol (**canFinalize**).

- **reserved1**
  This 3 byte field is used for alignment to a four-byte boundary.

- **colorServices**
  This field contains a pointer to the **ColorServices** callback documented in the general documentation. It contains null if the callback is not supported.

- **advanceState**
  The **advanceState** callback allows one to drive the interaction through the inner (**acquireSelectorContinue**) loop without actually returning from the plug-in. If it returns an error, then the plug-in generally should treat this as a continue error and pass it on when it returns.

- **reserved**
  These are set to zero by the host for future expansion of the plug-in standard. Must not be used by plug-ins.

Calling Order
When the user invokes the plug-in by selecting its name from the Acquire submenu, Photoshop loads the plug-in's resource into memory and calls it with the following sequence of selector values (see the header file for their actual values):

(1) acquireSelectorPrepare
This allows the plug-in to adjust Photoshop's memory allocation algorithm. Before this call, Photoshop sets **maxData** to the maximum number of bytes it would be able to free up. The plug-in module then has the option of reducing this number during this call. Reducing the number can speed up operation, since freeing up the maximum amount of memory requires Photoshop to move all of the image data for any currently open images out of of RAM and into its virtual memory file. Furthermore, in order to keep this amount of memory free, Photoshop is required to write any newly acquired image data to the virtual memory file as it is received.
If the plug-in knows that its memory requirements will be limited (if it can return the image data in strips, or if the maximum resolution image it can return is small), it should reduce **maxData** to its actual requirements during this call. This will allow small acquisitions to be performed entirely in RAM.

If, as is often the case, the plug-in only needs a small amount of memory, but will operate faster if given more, a tradeoff has to be made. One solution is to divide the **maxData** field by 2, thus allocating half the memory to Photoshop and half to the plug-in.
Another option is to reduce **maxData** to zero and then use the buffer and handle suites to allocate memory.

(2) acquireSelectorStart
This call lets Photoshop know the mode, size and resolution of the image being returned, so it can allocate and initialize its data structures. Most plug-ins will display their dialog box, if any, during this call.
During this call, the plug-in module should set **imageMode**, **imageSize**, **depth**, **planes**, **imageHRes** and **imageVRes**. If an indexed color image is being returned, it should also set **redLUT**, **greenLUT** and **blueLUT**. If a duotone mode image is being returned, it should also set **duotoneInfo**. See the descriptions of the fields given above.

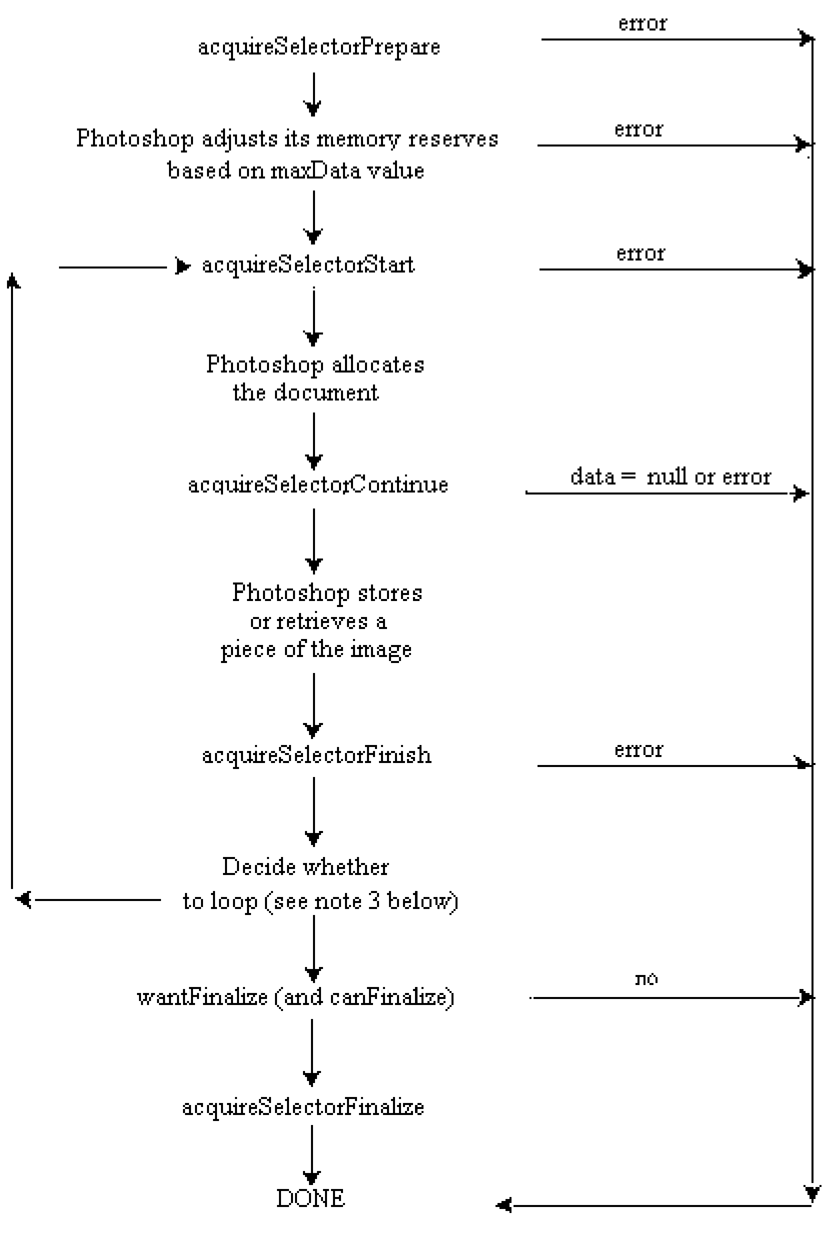
(3) acquireSelectorContinue
This call returns an area of the image to Photoshop. Photoshop will continue to call this routine until it either returns an error, or sets the **data** field to NULL.
The area of the image being returned is specified by **theRect** and by the **loPlane** and **hiPlane**. **data** contains a pointer to the actual data being returned. **colBytes**, **rowBytes** and **planeBytes** specify the organization of the data.
Photoshop is very flexible in the format in which image data can be returned. For example, to return just the red plane of an RGB color image, **loPlane** and the **hiPlane** should be set to 0, **colBytes** should be set to 1, and **rowBytes** should be set to the width of the area being returned (**planeBytes** is ignored in this case, since **loPlane** = **hiPlane**).
If instead, you wish to return the RGB data in interleaved form (RGBRGB...), the **loPlane** should be set to 0, **hiPlane** to 2, **planeBytes** to 1, **colBytes** to 3, and **rowBytes** to 3 times the width of area being returned.
The portion of the image being returned is specified by **theRect**. If the resolution of the acquired image is always going to be very small (e.g., NTSC frame grabbers), the plug-in can simply set **theRect** to the entire image area. However, if you wish to be able to scan large images, the plug-in must use the **theRect** field to return the image pieces. There are no restrictions on how the pieces tile the image (i.e., horizontal and vertical strips are allowed as are a grid of tiles). Each piece should contain no more than **maxData** bytes (less the size of any large tables or scratch areas allocated by the plug-in) unless the buffer for the image data was allocated using the buffer or handle suites.
The data field contains a pointer to the data being returned. Most plug-ins will allocate a buffer for the data using the **NewPtr** trap (or **GlobalAlloc( )** on Windows) or via the buffer suite. The plug-in is responsible for freeing this buffer in the **acquireSelectorFinish** call. (Note: this is a change from pre-version 3 interfaces, which freed the block for the plug-in!)

(4) acquireSelectorFinish
This call allows the plug-in to clean up after an image acquisition. This call is made if and only if the **acquireSelectorStart** routine returns without error (even if the **acquireSelectorContinue** routine returns an error).
Most plug-ins will at least need to free the buffer used to return the image data.
If Photoshop detects a command-period while processing the results of **acquireSelectorContinue** call, it will call the **acquireSelectorFinish** routine (be careful here, since normally the plug-in would be expecting another **acquireSelectorContinue** call).

(5) acquireSelectorFinalize
If the plug-in is using finalization (i.e., the host set **canFinalize** and the plug-in set **wantFinalize**), then thiscall will be made after all possible looping is complete.

State Machine

Photoshop plug-in acquire module's state machine


acquireSelectorPrepare
error
Photoshop adjusts its memory reserves based on maxData value
error
acquireSelectorStart
error
Photoshop allocates the document
acquireSelectorContinue
data = null or error
Photoshop stores or retrieves a piece of the image
acquireSelectorFinish
error
Decide whether to loop (see note 3 below)
wantFinalize (and canFinalize)
no
acquireSelectorFinalize
DONE

Notes:
1. If **acquireSelectorPrepare** succeeds -- i.e. , the result value is zero -- and **wantFinalize** is TRUE, then Photoshop guarantees that **acquireSelectorFinish** will be called.

2. If **acquireSelectorStart** succeeds then Photoshop guarantees that **acquireSelectorFinish** will be called.
3. Photoshop supports multiple document acquire which allows us to loop back to **acquireSelectorStart**. The simplest looping occurs after a successful acquisition sequence which ends with **acquireAgain** set TRUE. If the plug-in is using finalization (i.e., the host set **canFinalize** and the plug-in set **wantFinalize**), then we can also loop if **acquireAgain** is TRUE and the error code which terminated acquistion was > 0 or equalled **userCanceledErr**.
4. In the event of any error during acquisition, the document being acquired is discarded.
5. Hosts may choose to just treat **acquireAgain** as FALSE.
6. The plug-in can tell whether the host understands finalization by checking the **canFinalize** flag.
7. The **advanceState** callback allows one to drive the interaction through the inner (**acquireSelectorContinue**) loop without actually returning from the plug-in. If it returns an error, then the plug-in generally should treat this as an error **acquireSelectorContinue** continue and pass it on when it returns.

Error return values

The plug-in module may also return standard operating system error codes, or report its own errors, in which case it can return any positive integer.

```
#define acquireBadParameters      -30000 an error with the interface
#define acquireNoScanner          -30001 no scanner installed
#define acquireScannerProblem     -30002 a problem with the scanner
```

Sample Plug-in

DummyScan

is a sample acquire module. This is a new version of DummyScan which is 3.0 specific since it uses advanceState and the improved multiple acquire design.

Export Modules

Basics

The code resource and file type for export modules is **'8BEM'**.

The ExportRecord Structure

The **stuff** parameter contains a pointer to a structure of the following type:

```
typedef struct ExportRecord
{
	int32			serialNumber;
	TestAbortProc		abortProc;
	ProgressProc		progressProc;
	int32			maxData;

	int16			imageMode;
	Point			imageSize;
	int16			depth;
	int16			planes;

	Fixed			imageHRes;
	Fixed			imageVRes;

	LookUpTable		redLUT;
	LookUpTable		greenLUT;
	LookUpTable		blueLUT;

	Rect			theRect;
	int16			loPlane;
	int16			hiPlane;

	void *			data;
	int32			rowBytes;

	Str255			fileName;
	int16			vRefNum;
	Boolean			dirty;

	Rect			selectBBox;

	OSType			hostSig;
	ProcPtr		hostProc;

	Handle			duotoneInfo;
```

```
        int16                   thePlane;

        PlugInMonitor           monitor;

        void *                  platformData;

        BufferProcs     *       bufferProcs;
        ResourceProcs *         resourceProcs;

        ProcessEventProc        processEvent;

        DisplayPixelsProc       displayPixels;

        HandleProcs *           handleProcs;
        ColorServicesProc       colorServices;

        GetPropertyProc         getProperty;
        AdvanceStateProc        advanceState;

        int16                   layerPlanes;
        int16                   transparencyMask;
        int16                   layerMasks;
        int16                   invertedLayerMasks;
        int16                   nonLayerPlanes;

        char                    reserved [210];

    } ExportRecord;
```

Record Fields

- **serialNumber**
  This field contains Adobe Photoshop's serial number. Plug-in modules can use this value for copy protection, if desired.

- **abortProc**
  This field contains a pointer to the **TestAbort** callback documented in the general documentation.

- **progressProc**
  This field contains a pointer to the **UpdateProgress** callback documented in the general documentation. This procedure should only be called during the actual main operation of the plug-in, not during long operations during the preliminary user interface.

- **maxData**
  Photoshop initializes this field to the maximum of number of bytes it can free up. The plug-in may reduce this value during the **exportSelectorPrepare** routine. The **exportSelectorContinue** routine

should process the image in pieces no larger than **maxData**, less the size of any large tables or scratch areas it has allocated.

- **imageMode**
  The mode of the image being exported (grayscale, RGB Color, etc.). See the header file for possible values. The **exportSelectorStart** should return an **exportBadMode** error if it is unable to process this mode of image.

- **imageSize**
  The image's width (**imageSize.h**) and height (**imageSize.v**) in pixels.

- **depth**
  The image's resolution in bits per pixel per plane. The only possible settings are 1 for bitmap mode images, and 8 for all other modes.

- **planes**
  The number of channels in the image. For example, if an RGB image without alpha channels is being processed, this field will be set to 3.

- **imageHRes**
- **imageVRes**
  The image's horizontal and vertical resolution in terms of pixels per inch. These are fixed point numbers (16 binary digits).

- **redLUT**
- **greenLUT**
- **blueLUT**
  If an indexed color or duotone mode image is being processed, these fields will contain its color table.

- **theRect**
  The **exportSelectorStart** and **exportSelectorContinue** routines should set this field to request a piece of the image for processing. It should be set to an empty rectangle when complete.

- **loPlane**
- **hiPlane**
  The **exportSelectorStart** and **exportSelectorContinue** routines should set these fields to the first and last planes to process next.

- **data**
  This field contains a pointer to the requested image data. If more than one plane has been requested (**loPlane** < > **hiPlane**), the data is interleaved.

- **rowBytes**
  The offset between rows for the requested image data.

- **fileName**

The name of the file the image was read from. File exporting modules should use this field as the default name for saving.

- **vRefNum**
  The volume reference number of the file the image was read from.

- **dirty**
  File exporting modules should clear this field to prevent the user being prompted to save any unsaved changes when the image is eventually closed.

- **selectBBox**
  The bounding box of the current selection. If there is no current selection, this is an empty rectangle.

- **hostSig**
  The host program's signature. Photoshop's signature is **'8BIM'**.

- **hostProc**
  If not null, this field contains a pointer to a host-defined callback procedure that can do anything the host wishes. Plug-ins should verify **hostSig** before calling this procedure. This provides a mechanism for hosts to extend the plug-in interface to support application specific features.

- **duotoneInfo**
  When exporting a duotone mode image, the host allocates a handle and fills it with the duotone information. The format of the information is the same as that required by acquisition modules, and should be treated as a black box by plug-ins.

- **thePlane**
  Currently selected channel, or -1 if a composite color channel, or -2 if some other combination of channels.

- **monitor**
  This field contains the monitor setup information for the host. See the general documentation for more details.

- **platformData**
  This field contains a pointer to platform specific data. Not used on the Macintosh.

- **bufferProcs**
  This field contains a pointer to the buffer suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **resourceProcs**
  This field contains a pointer to the pseudo-resource suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **processEvent**

This field contains a pointer to the **ProcessEvent** callback documented in the general documentation. It contains null if the callback is not supported.

- **displayPixels**
  This field contains a pointer to the **DisplayPixels** callback documented in the general documentation. It contains null if the callback is not supported.

- **handleProcs**
  This field contains a pointer to the handle suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **colorServices**
  This field contains a pointer to the **ColorServices** callback documented in the general documentation. It contains null if the callback is not supported.

- **getProperty**
  This field contains a pointer to the property suite if it is supported by the host, otherwise null. (See **getProperty** documentation).

- **advanceState**
  The advanceState callback allows one to drive the interaction through the inner (exportSelectorContinue) loop without actually returning from the plug-in. If it returns an error, then the plug-in generally should treat this as a continue error and pass it on when it returns.

For documents with transparency, the export module is passed the merged
data together with the layer mask for the current target layer. The
structure is documented in the following fields:

- **layerPlanes**
  This field contains the number of planes of data possibly governed by a transparency mask.

- **transparencyMask**
  This field contains 1 or 0 indicating whether the data is governed by a transparency mask.

- **layerMasks**
  This field contains the number of layers masks (currently 1 or 0) for which 255 = fully opaque.

- **invertedLayerMasks**
  This field contains the number of layers masks (currently 1 or 0) for which 255 = fully transparent.

- **nonLayerPlanes**
  This field contains the number of planes of non-layer data, e.g., flat data or alpha channels.
  The planes are arranged in that order. Thus, an RGB image with an alpha channel and a layer mask on the current target layer would appear as: red, green, blue, transparency, layer mask, alpha channel

- **reserved**

These bytes are set to zero by the host for future expansion of the plug-in standard. Must not be used by plug-ins.

Calling Order

When the user invokes the plug-in by selecting its name from the Export submenu, Photoshop loads the plug-in's resource into memory and calls it with the following sequence of selector values (see the header file for their actual values):

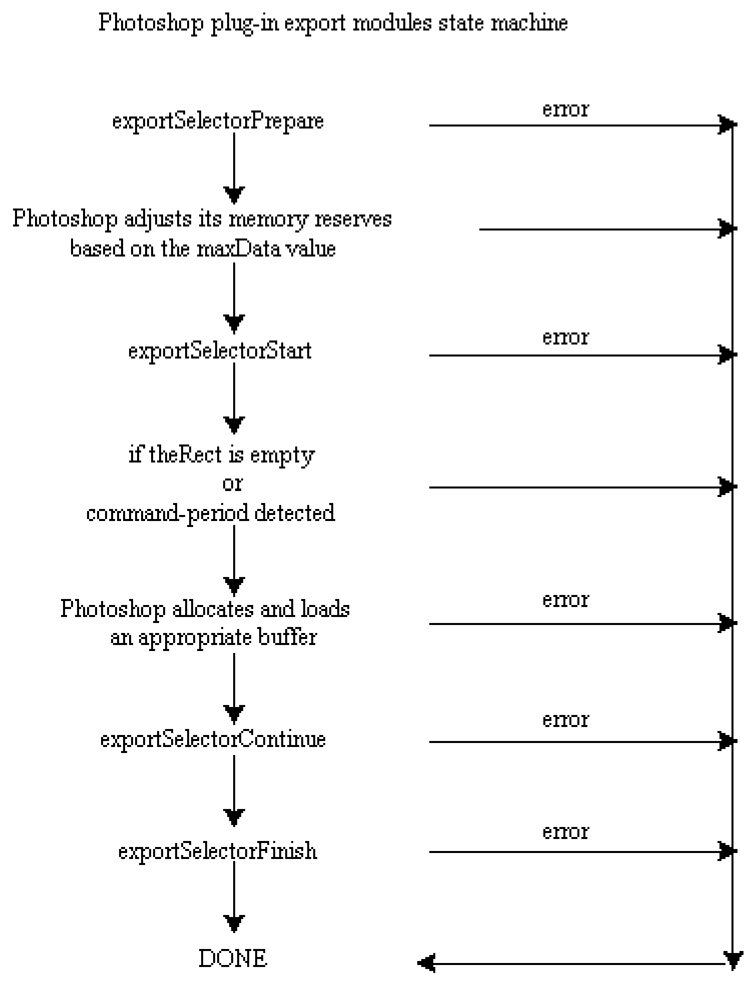
(1) exportSelectorPrepare
This allows the plug-in to adjust Photoshop's memory allocation algorithm. Before this call, Photoshop sets **maxData** to the maximum number of bytes it would be able to free up. The plug-in module then has the option of reducing this number during this call. Reducing the number can speed up operation, since freeing up the maximum amount of memory requires Photoshop to move all of the image data for any currently open images out of of RAM and into its virtual memory file.
If the plug-in knows that its memory requirements will be limited (if it can process the image data in strips, or if the maximum resolution image it can process is small), it should reduce **maxData** to its actual requirements during this call. This will allow small exports to be performed entirely in RAM.
If, as is often the case, the plug-in only needs a small amount of memory, but will operate faster if given more, a tradeoff has to be made. One solution is to divide the **maxData** field by 2, thus allocating half the memory to Photoshop and half to the plug-in.

(2) exportSelectorStart
Most plug-ins will display their dialog box, if any, during this call.
During this call, the plug-in should set **theRect**, **loPlane** and **hiPlane** to let Photoshop know what area of the image it wishes to process first.
The total number of bytes requested should be less than **maxData**. If the image is larger than **maxData**, the plug-in must process the image in pieces. There are no restrictions on how the pieces tile the image (i.e., horizontal and vertical strips are allowed as are a grid of tiles).

(3) exportSelectorContinue
During this routine, the plug-in should process the image data pointed to by data. It should then adjust **theRect**, **loPlane** and **hiPlane** to let Photoshop know what area of the image it wishes to process next. If the entire image has been processed, it should set **theRect** to an empty rectangle.
The requested image data is pointed to by data. If more than one plane has been requested (**loPlane** < > **hiPlane**), the data is interleaved. The offset from one row to the next is indicated by rowBytes. This is not necessarily equal to the width of **theRect** - there may be additional pad bytes at the end of each row!

(4) exportSelectorFinish
This call allows the plug-in to clean up after an image export. This call is made if and only if the **exportSelectorStart** routine returns without error (even if the **exportSelectorContinue** routine returns an error).
If Photoshop detects a command-period between calls to the **exportSelectorContinue** routine, it will call the **exportSelectorFinish** routine (be careful here, since normally the plug-in would be expecting another **exportSelectorContinue** call).

State Machine


Photoshop plug-in export modules state machine
exportSelectorPrepare
error
Photoshop adjusts its memory reserves based on the maxData value
exportSelectorStart
error
if theRect is empty or command-period detected
Photoshop allocates and loads an appropriate buffer
error
exportSelectorContinue
error
exportSelectorFinish
error
DONE

Notes:

1. If exportSelectorStart succeeds then Photoshop guarantees that exportSelectorFinish will be called.
2. Photoshop may choose to go exportSelectorFinish instead of exportSelectorContinue if it detects a need to terminate while building the requested buffer.
3. advanceState can be called from either exportSelectorStart or exportSelectorContinue and will drive Photoshop through the process of allocating and loading the requested buffer. Termination is reported as userCanceledErr in the result from the advanceState call. Calling advanceState when theRect is empty will result in no work being done.

Error return values

The plug-in module may also return standard operating system error codes, or report its own errors, in which case it can return any positive integer.

#define exportBadParameters	-30200	an error with the interface parameters
#define exportBadMode	-30201	the module does not support <mode> images

Sample Plug-ins

DummyExport

is a sample export module.

HistoryExport

is a sample export module primarily concerned with demonstrating the pseudo-resource callbacks. It works in conjunction with the Dissolve plug-in to maintain a series of history strings for a file. Not applicable for the Windows platform.

Paths to Illustrator

demonstrates using the getProperties callback and exporting of pen path information. The sample code works only on Macintosh platforms. It is fairly straightforward to extend the porting concepts from other examples to port this one over to the Windows platform. Please read the comments inside the sample source for important information regarding pen paths (like byte ordering etc.).

Filter Modules

Basics

The code resource and file type for filter modules is **'8BFM'**.

The FilterRecord Structure

The stuff parameter contains a pointer to a structure of the following type:

```
typedef char FilterColor [4];

typedef struct FilterRecord
{
        int32               serialNumber;

        TestAbortProc       abortProc;
        ProgressProc        progressProc;

        Handle              parameters;

        Point               imageSize;

        int16               planes;

        Rect                filterRect;

        RGBColor            background;
        RGBColor            foreground;

        int32               maxSpace;
        int32               bufferSpace;

        Rect                inRect;

        int16               inLoPlane;
        int16               inHiPlane;

        Rect                outRect;

        int16               outLoPlane;
        int16               outHiPlane;

        Ptr                 inData;
        int32               inRowBytes;

        Ptr                 outData;
```

```
int32                   outRowBytes;

Boolean                 isFloating;

Boolean                 haveMask;
Boolean                 autoMask;

Rect                    maskRect;

Ptr                     maskData;
int32                   maskRowBytes;

FilterColor             backColor;
FilterColor             foreColor;

OSType                  hostSig;
ProcPtr             hostProc;

int16                   imageMode;

Fixed                   imageHRes;
Fixed                   imageVRes;

Point                   floatCoord;
Point                   wholeSize;

PlugInMonitor           monitor;

void *                  platformData;

BufferProcs *           bufferProcs;
ResourceProcs *         resourceProcs;

ProcessEventProc        processEvent;

DisplayPixelsProc       displayPixels;

HandleProcs *           handleProcs;

Boolean                 supportsDummyPlanes;

Boolean                 supportsAlternateLayouts;

int16                   wantLayout;

int16                   filterCase;
```

```
int16                   dummyPlaneValue;

void *                  premiereHook;

AdvanceStateProc        advanceState;

Boolean                 supportsAbsolute;

Boolean                 wantsAbsolute;

GetPropertyProc         getProperty;

Boolean                 cannotUndo;

Boolean                 supportsPadding;

int16                   inputPadding;

int16                   outputPadding;

int16                   maskPadding;

char                    samplingSupport;

char                    reservedByte;

Fixed                   inputRate;
Fixed                   maskRate;

ColorServicesProc       colorServices;

int16                   inLayerPlanes;
int16                   inTransparencyMask;
int16                   inLayerMasks;
int16                   inInvertedLayerMasks;
int16                   inNonLayerPlanes;

int16                   outLayerPlanes;
int16                   outTransparencyMask;
int16                   outLayerMasks;
int16                   outInvertedLayerMasks;
int16                   outNonLayerPlanes;

int16                   absLayerPlanes;
int16                   absTransparencyMask;
```

```
        int16                   absLayerMasks;
        int16                   absInvertedLayerMasks;
        int16                   absNonLayerPlanes;


        int16                   inPreDummyPlanes;
        int16                   inPostDummyPlanes;

        int16                   outPreDummyPlanes
        int16                   outPostDummyPlanes;


        int32                   inColumnBytes;
        int32                   inPlaneBytes

        int32                   outColumnBytes;
        int32                   outPlaneBytes;

        char                    reserved [134];
    } FilterRecord;
```

Record Fields

- **serialNumber**
  This field contains Adobe Photoshop's serial number. Plug-in modules can use this value for copy protection, if desired.

- **abortProc**
  This field contains a pointer to the **TestAbort** callback documented in the general documentation.

- **progressProc**
  This field contains a pointer to the **UpdateProgress** callback documented in the general documentation. This procedure should only be called during the actual main operation of the plug-in, not during long operations during the preliminary user interface such as building a preview.

- **parameters**
  Photoshop initializes this handle to NULL at startup. If a plug-in filter has any parameters that the user can set, it should allocate a relocatable block in the **filterSelectorParameters** routine, store the parameters in the block, and store the block's handle in this field.

- **imageSize**
  The image's width (**imageSize.h**) and height (**imageSize.v**) in pixels. If the selection is floating, this field instead holds the size of the floating selection.

- **planes**

For version 4 filters, this field contains the total number of active planes in the image, including alpha channels. The image mode should be determined by looking at **imageMode**. For pre-version 4 filters, this field will be equal to 3 if filtering the RGB "channel" of an RGB color image, or 4 if filtering the CMYK "channel" of a CMYK color image. Otherwise it will be equal to 1.

- **filterRect**
  The area of the image to be filtered. This is the bounding box of the selection, or if there is no selection, the bounding box of the image. If the selection is not a perfect rectangle, Photoshop automatically masks the changes to the area actually selected (unless the plug-in turns off this feature using **autoMask**). This allows most filters to ignore the selection mask, and still operate correctly.

- **background**
- **foreground**
  The current background and foreground colors. If planes is equal to 1, these will have already been converted to monochrome. (Obsolete: Use **backColor** and **foreColor**.)

- **maxSpace**
  This lets the plug-in know the maximum number of bytes of information it can expect to be able to access at once (input area size + output area size + mask area size + **bufferSpace**).

- **bufferSpace**
  If the plug-in is planning on allocating any large internal buffers or tables, it should set this field during the **filterSelectorPrepare** call to the number of bytes it is planning to allocate. Photoshop will then try to free up the requested amount of space before calling the **filterSelectorStart** routine.

- **inRect**
  The plug-in should set this field in its **filterSelectorStart** and **filterSelectorContinue** routines to request access to an area of the input image. The area requested must be a subset of the image's bounding rectangle. After the entire **filterRect** has been filtered, this field should be set to an empty rectangle.

- **inLoPlane**
- **inHiPlane**
  The **filterSelectorStart** and **filterSelectorContinue** routines should set these fields to the first and last input planes to process next.

- **outRect**
  The plug-in should set this field in its **filterSelectorStart** and **filterSelectorContinue** routines to request access to an area of the output image. The area requested must be a subset of **filterRect**. After the entire **filterRect** has been filtered, this field should be set to an empty rectangle.

- **outLoPlane**
- **outHiPlane**
  The **filterSelectorStart** and **filterSelectorContinue** routines should set these fields to the first and last output planes to process next.

- **inData**

This field contains a pointer to the requested input image data. If more than one plane has been requested (**inLoPlane** < > **inHiPlane**), the data is interleaved.

- **inRowBytes**
  The offset between rows of the input image data. (There may or may not be pad bytes at the end of each row.)

- **outData**
  This field contains a pointer to the requested output image data. If more than one plane has been requested (**outLoPlane** < > **outHiPlane**), the data is interleaved.

- **outRowBytes**
  The offset between rows of the output image data. (There may or may not be pad bytes at the end of each row.)

- **isFloating**
  This field is set true if and only if the selection is floating.

- **haveMask**
  This field is set true if and only if a non-rectangular area has been selected.

- **autoMask**
  By default, Photoshop automatically masks any changes to the area actually selected. If **isFloating** is false, and **haveMask** is true, the plug-in can turn off this feature by setting this field to false. It can then perform its own masking.

- **maskRect**
  If **haveMask** is true, and the plug-in needs access to the selection mask, it should set this field in its **filterSelectorStart** and **filterSelectorContinue** routines to request access to an area of the selection mask. The requested area must be a subset of **filterRect**. This field is ignored if there is no selection mask.

- **maskData**
  A pointer to the requested mask data.

- **maskRowBytes**
  The offset between rows of the mask data.

- **backColor**
- **foreColor**
  The current background and foreground colors, in the color space native to the image.

- **hostSig**
  The host program's signature. Photoshop's signature is '**8BIM**'.

- **hostProc**

If not null, this field contains a pointer to a host-defined callback procedure that can do anything the host wishes. Plug-ins should verify **hostSig** before calling this procedure. This provides a mechanism for hosts to extend the plug-in interface to support application specific features.

- **imageMode**
  The mode of the image being filtered (Gray Scale, RGB Color, etc.). See the header file for possible values. The **filterSelectorStart** should return a **filterBadMode** error if it is unable to process this mode of image.

- **imageHRes**
- **imageVRes**
  The image's horizontal and vertical resolution in terms of pixels per inch. These are fixed point numbers (16 binary digits).

- **floatCoord**
  If **isFloating** is true, the coordinate of the top-left corner of the floating selection in the main image's coordinate space.

- **wholeSize**
  If **isFloating** is true, the size in pixels of the entire main image.

- **monitor**
  This field contains the monitor setup information for the host. See the general documentation for more details.

- **platformData**
  This field contains a pointer to platform specific data. Not used on the Macintosh.

- **bufferProcs**
  This field contains a pointer to the buffer suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **resourceProcs**
  This field contains a pointer to the pseudo-resource suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **processEvent**
  This field contains a pointer to the **ProcessEvent** callback documented in the general documentation. It contains null if the callback is not supported.

- **displayPixels**
  This field contains a pointer to the **DisplayPixels** callback documented in the general documentation. It contains null if the callback is not supported.

- **handleProcs**

This field contains a pointer to the handle suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **supportsDummyPlanes**
  Does the host support the plug-in requesting non-existent planes? (see dummy planes fields below) This field is set by the host to indicate whether it respects the dummy planes fields.

- **supportsAlternateLayouts**
  Does the host support data layouts other than rows of columns of planes? This field is set by the host to indicate whether it respects the **wantLayout** field.

- **wantLayout**
  The desired layout for the data. See **PIGeneral.h**. The host only looks at this field if it has also set **supportsAlternateLayouts**.

- **filterCase**
  The type of data being filtered, flat, floating, layer with editable transparency, layer with preserved transparency. With and without a selection. A zero indicates that the host did not set this field.

- **dummyPlaneValue**
  The value to store into any dummy planes. 0..255 = specific value. -1 = leave undefined (i.e., random)

- **premiereHook**
  See the Adobe Premiere™ Plug-in Developer's Kit.

- **advanceState**
  See above.

- **supportsAbsolute**
  Does the host support absolute channel indexing? Absolute channel indexing ignores visiblity concerns and numbers the channels from zero starting with the first composite channel if any, followed by the transparency, followed by any layer masks, followed by any alpha channels.

- **wantsAbsolute**
  Enable absolute channel indexing for the input. This is only useful if **supportsAbsolute** is true. Absolute indexing is useful for things like accessing alpha channels.

- **getProperty**
  See **getProperty** in the general documentation section.

- **cannotUndo**
  If the filter makes a non-undoable change, then setting this field will prevent Photoshop from offering undo for the filter. This is rarely needed.

- **inputPadding**
- **outputPadding**

- **maskPadding**
  The input, output, and mask can be padded when loaded. The options for padding include specifying a specific value (0..255), specifying edge replication (plugInWantsEdgeReplication), specifying that the data be left random (plugInDoesNotWantPadding), or requesting that an error be signaled for an out of bounds request (plugInWantsErrorOnBoundsException). The error case is the default since previous versions would have errored out in this event.

- **samplingSupport**
  Does the host support non-1:1 sampling of the input and mask? Photoshop 3.0.1 supports integral sampling steps (it will round up to get there). This is indicated by the value hostSupportsIntegralSampling. Future versions may support non-integral sampling steps. This will be indicated with *hostSupportsFractionalSampling*.

- **inputRate**
  The sampling rate for the input. The effective input rectangle (in normal sampling coordinates) is inRect * inputRate (i.e., inRect.top * inputRate, inRect.left * inputRate, inRect.bottom * inputRate, inRect.right * inputRect). inputRate is rounded to the nearest integer in Photoshop 3.0.1. Since the scaled rectangle may exceed the real source data, it is a good idea to set some sort of padding for the input as well.

- **maskRate**
  Like inputRate, but as applied to the mask data.

- **inLayerPlanes**
- **inTransparencyMask**
- **inLayerMasks**
- **inInvertedLayerMasks**
- **inNonLayerPlanes**
  The number of planes (channels) in each category for the input data. This is the order in which the planes are presented to the plug-in and as such gives the structure of the input data. The inverted layer masks are ones where 0 = fully visible and 255 = completely hidden. If these are all zero, then the plug-in should assume the host has not set them.

- **outLayerPlanes**
- **outTransparencyMask**
- **outLayerMasks**
- **outInvertedLayerMasks**
- **outNonLayerPlanes**
  The structure of the output data. This will be a prefix of the input planes. For example, in the protected transparency case, the input can contain a transparency mask and a layer mask while the output will contain just the layerPlanes.

- **absLayerPlanes**
- **absTransparencyMask**
- **absLayerMasks**
- **absInvertedLayerMasks**
- **absNonLayerPlanes**

The structure of the input data when wantsAbsolute is true.

- **inPreDummyPlanes**
- **inPostDummyPlanes**
  The number of extra planes before and after the input data. This is only available if supportsDummyChannels is TRUE. This is used for things like forcing RGB data to appear as RGBA.

- **outPreDummyPlanes**
- **outPostDummyPlanes**
  Like inPreDummyPlanes & inPostDummyPlanes except it applies to the output data.

- **inColumnBytes**
  The step from column to column in the input. If using the layout options, this value may change from being equal to the number of planes. If it is zero, the plug-in should assume that the host has not set it.

- **inPlaneBytes**
  The step from plane to plane in the input. Normally one, but this changes if the plug-in uses the layout options. If it is zero, the plug-in should assume that the host has not set it.

- **outColumnBytes**
- **outPlaneBytes**
  The output equivalent of the previous two fields.

- **reserved**
  These bytes are set to zero by the host for future expansion of the plug-in standard. Must not be used by plug-ins.

Calling Order

When the user invokes the plug-in by selecting its name from the Filter submenu, Photoshop loads the plug-in's resource into memory and calls it with the following sequence of selector values (see the header file for their actual values):

(1) filterSelectorParameters
If the plug-in filter has any parameters that the user can set, it should prompt the user and save the parameters in a relocatable memory block whose handle is stored in the parameters field. Photoshop initializes the parameters field to **NULL** when starting up.
This routine may or may not be called depending on how the user invokes the filter. Photoshop has a feature the repeats the most recent filtering operation using the current parameters, in which case this call is skipped.
Since the same parameters can be used on different size images, the parameters should not depend on the size or mode of the image, or the size of the filtered area (these fields are not even defined at this point).
A future version of Photoshop may have a macro processor which would save the block pointed to by the parameters field when recording, so that it can operate the filter without user input during play back. Adobe Premiere[a] actually uses this feature. To be compatible with this feature, all parameters must be saved in a relocatable block whose handle is stored in the parameters field.
Ideally, the parameter block should contain the following information:

1. A signature so that the plug-in can do a quick confirmation that this is, in fact, one of its parameter blocks.
2. A version number so that the plug-in can evolve without requiring a new signature.
3. A convention regarding byte-order for cross-platform support (or a flag to indicate what byte order is being used).

The plug-in should validate the contents of its parameter handle when it starts processing if there is a danger of it crashing from bad parameters.

One other feature which can be put into plug-ins with respect to parameters is to store a default parameter handle in the plug-in's resource fork. This way, you can save preference settings for the plug-in across invocations of the host.

(2) filterSelectorPrepare

If the plug-in is planning on allocating any large (>= about 32K) buffers or tables, it should set the bufferSpace field to the number of bytes it is planning to allocate. Photoshop will then try to free up that amount of memory before calling the plug-in's **filterSelectorStart** routine. Alternatively, one can set this field to zero and use the buffer and handle suites if they are available.

The fields such as **imageSize**, **planes**, **filterRect**, etc. have now been defined, and can be used in computing the buffer size requirements.

(3) filterSelectorStart

The plug-in should set **inRect** and **outRect** (and **maskRect**, if it is using the selection mask) to request the first areas of the image to work on.

If at all possible, the plug-in should process the image in pieces to minimize memory requirements. Unless there is a lot of startup/shutdown overhead on each call (e.g., talking to an external DSP), tiling the image with rectangles measuring 64x64 to 128x128 seems to work fairly well.

(4) filterSelectorContinue

This routine is repeatedly called as long as at least one of the **inRect** , **outRect**, or **maskRect** fields is non-empty.

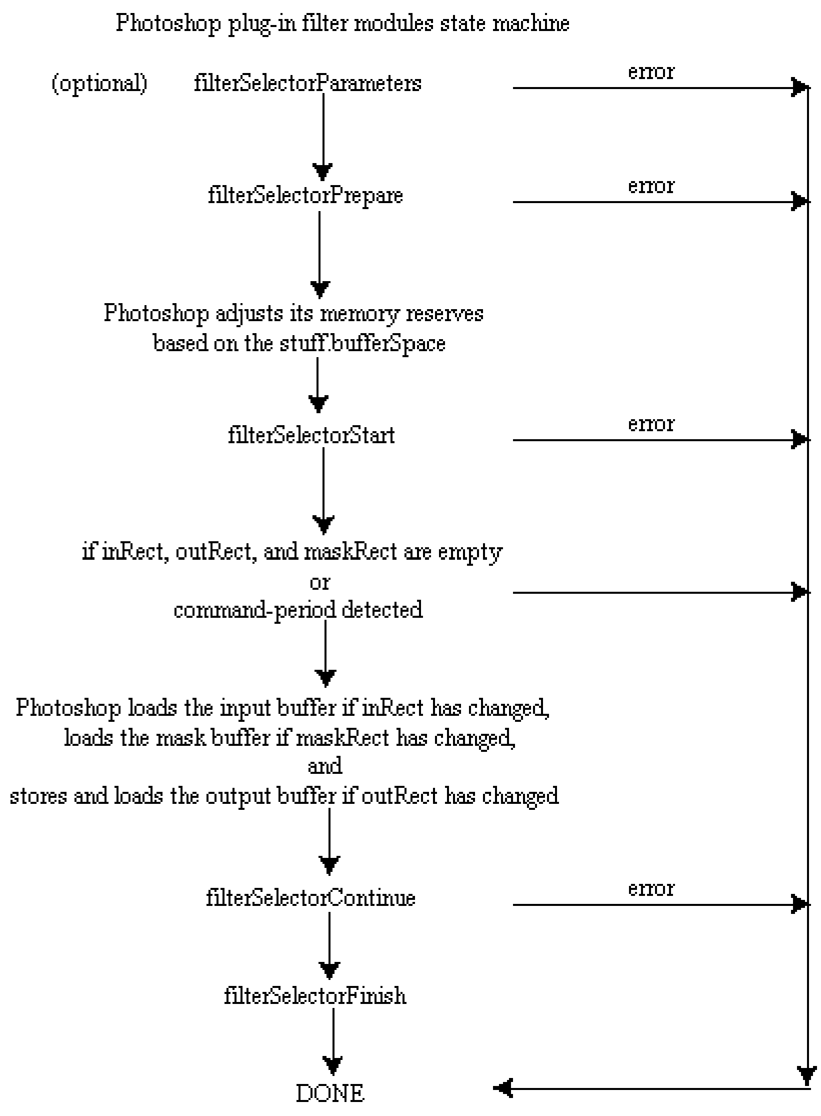
This routine should process the data pointed by **inData** and **outData** (and possibly **maskData**) and then update **inRect** and **outRect** (and **maskRect**, if using the selection mask) to request the next area of the image to process.

(5) filterSelectorFinish

This call allows the plug-in to clean up after a filtering operation. This call is made if and only if the **filterSelectorStart** routine returns without error (even if the **filterSelectorContinue** routine returns an error).

If Photoshop detects a command-period (i.e. user presses the Escape Key on Windows) between calls to the **filterSelectorContinue** routine, it will call the **filterSelectorFinish** routine (be careful here, since normally the plug-in would be expecting another filterSelectorContinue call).

State Machine


Photoshop plug-in filter modules state machine
(optional)
filterSelectorParameters
error
filterSelectorPrepare
error
Photoshop adjusts its memory reserves based on the stuff.bufferSpace
filterSelectorStart
error
if inRect, outRect, and maskRect are empty or command-period detected
Photoshop loads the input buffer if inRect has changed, loads the mask buffer if maskRect has changed, and stores and loads the output buffer if outRect has changed
filterSelectorContinue
error
filterSelectorFinish
DONE

Notes:

1. If **filterSelectorStart** succeeds, then Photoshop guarantees that **filterSelectorFinish** will be called.
2. Photoshop may choose to go to **filterSelectorFinish** instead of **filterSelectorContinue** if it detects a need to terminate while fulfilling a request.
3. **AdvanceState** may be called from either **filterSelectorStart** or **filterSelectorFinish** and will drive Photoshop through the buffer set up code. If the rectangles are empty, the buffers will simply be cleared. Termination is reported as **userCanceledErr** in the result from the **advanceState** call.

Error return values

The plug-in module may also return standard operating system error codes, or report its own errors, in which case it can return any positive integer.

#define filterBadParameters	-30100	a problem with the interface
#define filterBadMode	-30101	module doesn't support <mode> images

Sample Plug-in

Dissolve
is a sample filter plug-in which demonstrates layers.

Image Format Modules

Basics

The code resource and file type for acquisition modules is '**8BIF**'.

The FormatRecord Structure

The **stuff** parameter contains a pointer to a structure of the following type:

```
typedef struct FormatRecord
        {

        int32                   serialNumber;
        TestAbortProc           abortProc;
        ProgressProc            progressProc;
        int32                   maxData;

        int32                   minDataBytes;
        int32                   maxDataBytes;
        int32                   minRsrcBytes;
        int32                   maxRsrcBytes;

        int32                   dataFork;
        int32                   rsrcFork;

        int16                   imageMode;
        Point                   imageSize;
        int16                   depth;
        int16                   planes;

        Fixed                   imageHRes;
        Fixed                   imageVRes;

        LookUpTable             redLUT;
        LookUpTable             greenLUT;
        LookUpTable             blueLUT;

        void *                  data;

        Rect                    theRect;
        int16                   loPlane;
        int16                   hiPlane;
        int16                   colBytes;
        int32                   rowBytes;
        int32                   planeBytes;
```

```
    int16                   planeMap [16];

    Boolean                 canTranspose;
    Boolean                 needTranspose;

    OSType                  hostSig;
    ProcPtr                 hostProc;

    int32                   hostModes;

    Handle                  revertInfo;

    NewPIHandleProc         newHandleProc;
    DisposePIHandleProc     disposeHandleProc;

    Handle                  imageRsrcData;
    int32                   imageRsrcSize;

    PlugInMonitor           monitor;

    void *                  platformData;

    BufferProcs *           bufferProcs;
    ResourceProcs *         resourceProcs;

    ProcessEventProc        processEvent;

    DisplayPixelsProc       displayPixels;

    HandleProcs *           handleProcs;

    OSType                  fileType;

    ColorServicesProc       colorServices

    AdvanceStateProc        advanceState;

    char                    reserved [236]; /* Set to zero */

} FormatRecord;
```

Image Resources

Photoshop documents can have other properties associated with them besides pixel data. For example, we save page setup information and pen tool paths. Photoshop supports the concept of a block of data known as the image resources for a file. Image formats can store and retrieve this information if the file format definition allows for a place to put such an arbitrary block of data (e.g., a TIFF tag or a PicComment).

Record Fields

- **serialNumber**
  This field contains Adobe Photoshop's serial number. Plug-in modules can use this value for copy protection, if desired.

- **abortProc**
  This field contains a pointer to the **TestAbort** callback documented in the general documentation.

- **progressProc**
  This field contains a pointer to the **UpdateProgress** callback documented in the general documentation. This procedure should only be called during the actual main operation of the plug-in, not during long operations during the preliminary user interface.

- **maxData**
  Photoshop initializes this field to the maximum of number of bytes it can free up. The plug-in may reduce this value during the prepare routines. The continue routines should process the image in pieces no larger than maxData less the size of any large tables or scratch areas it has allocated.

- **minDataBytes**
- **maxDataBytes**
  These fields give the limits on the data fork space needed to write the file. The plug-in should set these during the estimate sequence of selector calls.

- **minRsrcBytes**
- **maxRsrcBytes**
  These fields give the limits on the resource fork space needed to write the file. The plug-in should set these during the estimate sequence of selector calls.

- **dataFork**
  The reference number for the data fork of the file to be read during the read sequence or written during the write sequence. During the options and estimate sequences, this field is undefined. On Windows, this is the file handle of the file returned by OpenFile ( ) API.

- **rsrcFork**
  The reference number for the resource fork of the file to be read during the read sequence or written during the write sequence. During the options and estimate sequences, this field is undefined. On Windows, this field is undefined.

- **imageMode**
  The **formatSelectorReadStart** routine should set this field to inform Photoshop what mode image is being acquired (grayscale, RGB Color, etc.). See the header file for possible values. Photoshop will set this field before it calls **formatSelectorOptionsStart**, **formatSelectorEstimateStart**, or **formatSelectorWriteStart**.

- **imageSize**
  The **formatSelectorReadStart** routine should set this field to inform Photoshop of the image's width (**imageSize.h**) and height (**imageSize.v**) in pixels. Photoshop will set this field before it calls **formatSelectorOptionsStart**, **formatSelectorEstimateStart**, or **formatSelectorWriteStart**.

- **depth**
  The **formatSelectorReadStart** routine should set this field to inform Photoshop of the image's resolution in bits per pixel per plane. The only valid settings are 1 for bitmap mode images, and 8 for all other modes. Photoshop will set this field before it calls **formatSelectorOptionsStart**, **formatSelectorEstimateStart**, or **formatSelectorWriteStart**.

- **planes**
  The **formatSelectorReadStart** routine should set this field to inform Photoshop of the number of channels in the image. For example, if an RGB image without alpha channels is being returned, this field should be set to 3. Photoshop will set this field before it calls **formatSelectorOptionsStart**, **formatSelectorEstimateStart**, or **formatSelectorWriteStart**. Because of the implementation of the plane map, format modules (and acquire modules) should never try to work with more than 16 planes at a time. The results would be unpredictable.

- **imageHRes**
- **imageVRes**
  The **formatSelectorReadStart** routine should set these fields to inform Photoshop of the image's horizontal and vertical resolution in terms of pixels per inch. This is a fixed point number (16 binary digits). Photoshop initializes these fields to 72 pixels per inch. Photoshop will set these fields before it calls **formatSelectorOptionsStart**, **formatSelectorEstimateStart**, or **formatSelectorWriteStart**. The current version of Photoshop only supports square pixels, so it ignores the **imageVRes** field. Plug-ins should set both fields anyway in case future versions of Photoshop support non-square pixels.

- **redLUT**
- **greenLUT**
- **blueLUT**
  If an indexed color mode image is being returned, the formatSelectorReadStart routine should return the image's color table in these fields. If an indexed color document is being written, Photoshop will set these fields before it calls formatSelectorOptionsStart, formatSelectorEstimateStart, or formatSelectorWriteStart.

- **data**
  The start and continue routines should return a pointer to the buffer where image data is or is to be stored in this field. After the entire image has been processed, the continue selectors should set this field to NULL. Note that the plug-in is responsible for freeing any memory pointed to by this field.

- **theRect**
  The plug-in should set this to the area of the image covered by the buffer specified in data.

- **loPlane**
- **hiPlane**

The start and continue routines should set this to the first and last planes covered by the buffer specified in data. For example, if interleaved RGB data is being used, they should be set to 0 and 2, respectively.

- **colBytes**
  The start and continue routines should set this field to the offset in bytes between columns of data in the buffer. This is usually 1 for non-interleaved data, or (**hiPlane** - **loPlane** + 1) for interleaved data.

- **rowBytes**
  The start and continue routines should set this field to the offset in bytes between rows of data in the buffer.

- **planeBytes**
  The start and continue routines should set this field to the offset in bytes between planes of data in the buffers. This field is ignored if **loPlane** = **hiPlane**. It should be set to 1 for interleaved data.

- **planeMap**
  This is initialized by the host to a linear map (**planeMap [i]** = i). This is used to map plane (channel) numbers between the plug-in and the host. For example, Photoshop stores RGB images with an alpha channel in the order RGBA, whereas most frame buffers store the data in ARGB order. To work with the data in this order, the plug-in should set **planeMap [0]** to 3, **planeMap [1]** to 0, **planeMap [2]** to 1, and **planeMap [3]** to 2.

- **canTranspose**
  If the host supports transposing images during or after reading or before or during writing, it should set this field to true. Photoshop always sets this field to true.

- **needTranspose**
  Initialized by the host to false. If the plug-in wishes to have the image transposed, and **canTranspose** is true, it should set this field to true during the start call.

- **hostSig**
  The host program's signature. Photoshop's signature is '**8BIM**'.

- **hostProc**
  If not null, this field contains a pointer to a host-defined callback procedure that can do anything the host wishes. Plug-ins should verify **hostSig** before calling this procedure. This provides a mechanism for hosts to extend the plug-in interface to support application specific features.

- **hostModes**
  This field is used by the host to inform the plug-in which **imageMode** values it supports. If the corresponding bit (LSB = bit 0) is 1, the mode is supported. This field can be used by plug-ins to disable reading unsupported file formats.

- **revertInfo**
  This field is set to NULL by Photoshop when a format for a file is first created. If this field is defined on a **formatSelectorReadStart** call, then treat the call as a revert and don't query the user. If it is null on

the **formatSelectorReadStart** call, then query the user as appropriate and set up this field to store a handle containing the information necessary to read the file without querying the user for additional parameters (essential for reverting the file) and if possible to write the file without querying the user. The contents of this field are sticky to a document and will be duplicated when we duplicate the image format information for a document. On all **formatSelectorOptions** calls, leave **revertInfo** containing enough information to revert the document.
Photoshop will dispose of this field when it disposes of the document, hence, the plug-in must call on Photoshop to allocate the data as well using the following callbacks or the callbacks provided in the handle suite.

- **newHandleProc**
  This is the same as the **NewPIHandle** callback described in the general documentation. This field existed before the handle suite was defined.

- **disposeHandleProc**
  This is the same as the **DisposePIHandle** callback described in the general documentation. This field existed before the handle suite was defined.

- **imageRsrcData**
  During calls to the write sequence, this field contains a handle to a block of data to be stored in the file as image resource data. Since this handle is allocated before the write sequence begins, plug-ins must add any resources they want saved to the document during the options or estimate sequence. Since options is not always called, the best time is during the estimate sequence. During the read sequence, Photoshop checks this field after each call to **formatSelectorRead** and **formatSelectorContinue** and the first time it is non-NULL parses the handle as a block of image resource data for the current document.

- **imageRsrcSize**
  This is the size of the handle in **imageRsrcData**. It is really only relevant during the estimate sequence when it is provided instead of the actual resource data.

- **monitor**
  This field contains the monitor setup information for the host. See the general documentation for more details.

- **platformData**
  This field contains a pointer to platform specific data. Not used on the Macintosh.

- **bufferProcs**
  This field contains a pointer to the buffer suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **resourceProcs**
  This field contains a pointer to the pseudo-resource suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **processEvent**
  This field contains a pointer to the **ProcessEvent** callback documented in the general documentation. It contains null if the callback is not supported.

- **displayPixels**
  This field contains a pointer to the **DisplayPixels** callback documented in the general documentation. It contains null if the callback is not supported.

- **handleProcs**
  This field contains a pointer to the handle suite if it is supported by the host, otherwise null. (See the general plug-in documentation).

- **fileType**
  This field contains the file type for filtering.

- **colorServices**
  This field contains a pointer to the **ColorServices** callback documented in the general documentation. It contains null if the callback is not supported.

- **advanceState**
  The **advanceState** callback allows one to drive the interaction through the inner (**formatSelectorOptionsContinue**) loop without actually returning from the plug-in. If it returns an error, then the plug-in generally should treat this as an error **formatSelectorOptionsContinue** and pass it on when it returns.

- **reserved**
  Set to zero by the host for future expansion of the plug-in standard. Must not be used by plug-ins.

Calling Sequences

Image format plug-ins actually need to support a variety of selector calling sequences to support various pieces of the process of reading and writing files.

One sequence is used to read image files. It works much like acquire modules do and it should be relatively easy to convert acquire modules to read-only image formats.

Three sequences are used when writing a file. The first should be used to request save options from the user. It will only be used when first saving a file in a particular format. The second estimates the file size so that the host can decide whether there is enough disk space available. The last sequence actually writes the file.
All of these sequences follow the same general pattern:

(1) prepare
The prepare call is made to allow the plug-in to adjust Photoshop's memory allocation algorithm. Before this call, Photoshop sets maxData to the maximum number of bytes it would be able to free up. The plug-in module then has the option of reducing this number during this call. Reducing the number can speed up operation, since freeing up the maximum amount of memory requires Photoshop to move all of the image

data for any currently open images out of of RAM and into its virtual memory file. A smaller number will allow Photoshop to keep more image data in memory.
If the plug-in knows that its memory requirements will be limited (if it can return the image data in pieces, or if the biggest image it can return is small), it should reduce maxData to its actual requirements during this call. This will allow small acquisitions to be performed entirely in RAM.
If, as is often the case, the plug-in only needs a small amount of memory, but will operate faster if given more, a tradeoff has to be made. One solution is to divide **maxData** by 2, thus allocating half the memory to Photoshop and half to the plug-in.
If the plug-in can use the buffer and handle suites for all its memory allocation, this is even better. In this event, the plug-in should simply set **maxData** to zero.

(2) start
The start call allows the plug-in to begin its interaction with the host.
For formatSelectorReadStart, the plug-in should set **imageMode**, **imageSize**, **depth**, **planes**, **imageHRes** and **imageVRes**. If an indexed color image is being returned, it should also set **redLUT**, **greenLUT** and **blueLUT**. If the plug-in has a block of image resources it wishes processed it should set **imageRsrcData** to be a handle to the resource data.
For both reading and writing, the plug-in should set up the first input or output buffer as appropriate. The area of the image being returned or requested is specified by **theRect**, **loPlane**, and **hiPlane**. **data** contains a pointer to the actual pixels. **colBytes**, **rowBytes**, **planeBytes**, and **planeMap** specify the organization of the data.
For **formatSelectorReadStart** calls, the pixel data block should be filled in with the data to save. For **formatSelectorOptionsContinue**, **formatSelectorEstimateContinue**, and **formatSelectorWriteContinue** calls, the data block will be filled in with the requested pixel data at the beginning of the next call to the plug-in.
Photoshop is very flexible in the format in which image data can be transferred. For example, to return or request just the red plane of an RGB color image, **loPlane** and **hiPlane** should be set to 0, **colBytes** should be set to 1, and **rowBytes** should be set to the width of the area being returned (**planeBytes** is ignored in this case, since **loPlane** = **hiPlane**). If instead, you wish to return or request the RGB data in interleaved form (RGBRGB...), **loPlane** should be set to 0, **hiPlane** to 2, **planeBytes** to 1, **colBytes** to 3, and **rowBytes** to 3 times the width of area being returned.
The actual pixel data is stored in the block of memory pointed to by the contents of data. Most plug-ins will either use a **NewPtr** call or will allocate the memory using the buffer suite. The plug-in is responsible for freeing this memory during the finish call.

(3) continue
This call is used to process a sequence of areas within the image. The selected routine should process any incoming data and then, just as with the start call, set up **theRect**, **loPlane**, **hiPlane**, **planeMap**, **data**, **colBytes**, **rowBytes**, and **planeBytes** to describe the next chunk of the image being returned or requested. The host will keep calling the continue routine until data is NULL.

(4) finish
The finish selector allows the plug-in to clean-up from the operation just performed. This call is made if and only if the start call returns without error (even if one of the continue calls results in an error.)
Most plug-ins will at least need to free the buffer used to return or request pixel data if this has not been done previously.

If Photoshop detects a command-period while processing the results of a continue call, it will call the finish routine. Be careful here, since normally the plug-in would be expecting another continue call. This is why it is frequently best to do all of one's clean-up in the finish call.

Error return values

The plug-in module may also return standard operating system error codes, or report its own errors, in which case it can return any positive integer.

```
#define formatBadParameters      -30500 a problem with the module interface
#define formatCannotRead         -30501
```

Sample Plug-in

Sample Format

is a sample format module.

Document File Formats

Image Resource Block

The image resource data block stored in various files by Photoshop 3.0 is used to store non-pixel data which may be associated with an image such as pen tool paths. It is referred to as resource data because it holds data which would formerly have been stored in the resource fork of the file on the Macintosh.

It consists of successive blocks of data in the following format:

1. resource type (4 bytes, most often '**8BIM**')
2. resource ID (2 bytes)
3. resource name (a Pascal format string padded to make the size even)
4. resource size (4 bytes)
5. resource data (**resource size** bytes plus padding to make the size even)

Path Resource Format

Photoshop stores the paths saved with an image in the resource fork of the image file or in the image resource block. This document describes how to interpret and modify those paths.

(1) Photoshop stores its paths as resources of type '**8BIM**' with IDs in the range 2000 through 2998. Photoshop stores other information using resources of type '**8BIM**' so it is important to pay attention to the IDs. The name of the resource is the name given to the path when it was saved.

(2) If the file contains a resource of type '**8BIM**' with an ID of 2999, then this resource contains a Pascal-style string containing the name of the clipping path to use with this image when saving it as an EPS file.

Items 3 and 4 describe the path resource format in detail. The path format returned by GetProperty ( ) call is identical to what is described below (Please refer to Paths To Illustrator Sample).

(3) All points used in defining a path are stored as a pair of 32-bit components, vertical component first. The two components are fixed point numbers with 8 bits before the binary point and 24 bits after the binary point. We insist on leaving three guard bits in the points to eliminate most concerns over arithmetic overflow. Hence, the range for each component is 0xF0000000 to 0x0FFFFFFF representing a range of -16 to 16. We include the lower bound but not the upper bound. We use such a limited range because we express the points relative to the image size. The vertical component is given with respect to the image height and the horizontal component is given with respect to the image width. <0,0> represents the top-left corner of the image; <1,1> (<0x01000000,0x01000000>) represents the bottom-right. On a LittleEndian machine (Intel platform), the byte order is reversed. You should swap the bytes before accessing it as int32.

(4) The data in a path resource consists of a sequence of 26 byte records.

A. The first two bytes (bytes 0 and 1) of each record are a 16-bit value which indicates the kind of data contained in the rest of the record. On a LittleEndian Machine (Intel platform), you should swap the bytes before accessing it as a short (int16).

B. If the kind value is 0, 1, or 2, then this record is part of the description of a closed subpath within the compound path.

1. If the kind value is 0, then bytes 2 and 3 of the record contain the length of the closed subpath. Such a record is then followed by records describing the knots of the subpath. This must be the first record in the subpath description.

2. If the kind value is 1 or 2, then the remaining 24 bytes of the record represent three points in the above format giving the control point for the Bezier segment preceding the knot, the anchor point for the knot, and the control point for the Bezier segment leaving the knot in that order. If the kind value is 1, the control points are linked; i.e., editing one point edits the other one to preserve collinearity. Knots should only be marked as having linked controls if their control points are collinear with their anchor. If the kind value is 2, then this is a knot for which the control points are not linked.

C. If the kind value is 3, 4, or 5, then this record is part of the description of an open subpath within the compound path.

1. If the kind value is 3, then this is a path length record just like kind value 0.

2. If the kind value is 4, then this record contains the data for a knot with linked controls on the open subpath.

3. If the kind value is 5, then this record contains the data for a knot with non-linked control on the open subpath.

D. Further kind values may be added in the future. Since Photoshop will ignore records for which it does not understand the kind value, this is a relatively easy format to extend.

Photoshop 3.0

The Macintosh file type code is '**8BPS'**. The DOS extension is '.**PSD'**. All information is stored in big-endian byte order, so little-endian machines will have to swap bytes when reading and writing.

The 'File Info' in Photoshop 3.0 is stored in numerous places in a given file for various formats:

On the Mac the information is stored in the resource fork for any file format:

The keywords are stored in a 'KeyW' resource and the caption is stored in a 'TEXT' resource. These resources are referenced by the 'pnot' resource. This information is readable by Aldus Fetch. For more information on the format of these resources see:

Inside Macintosh: QuickTime Components
    and
Aldus Fetch Awareness Developer's Toolkit
  (206) 628-5693

All of the data from File Info is stored in an 'ANPA 1000' resource. The data in this resource is stored as an IPTC-NAA record 2 and should be readable by various tools from Iron Mike. For more information on the format of this resource see:

IPTC-NAA Digital Newsphoto Parameter Record
  Newspaper Association of America
  The Newspaper Center
  11600 Sunrise Valley Drive
  Reston VA 20091

On all platforms, the data is also stored in the data fork of the file. For file formats that can support Photoshop's image resources the data is stored as an image resource '1028' (kCaptionID) in IPTC-NAA record 2 format.

For TIFF files the caption data is stored in an image description tag '270' and all the information is stored as an IPTC-NAA record 2 in tag '33723' the tag number was chosen by inspecting files written by Iron Mike software, and is supposed to be defined in a Rich TIFF specification. The tag is also specified in:

NSK TIFF
  The Japan Newspaper Publishers & Editors Association
  Nippon Press Center Building
  2-2-1 Uchlsaiwai-cho
  Chiyoda-ku, Tokyo 100

For more information about the TIFF format see:

TIFF Revision 6.0

(206) 628-5693

In reading the files, the following order is used with information read lower on the list replacing information read higher.

Image Description Tag (TIFF only)
IPTC-NAA Tag (TIFF only)

kCaptionID image resource

For old Photoshop files, the caption data is read from the image resource '1008' (kOldCaptionID) or '1020' (kPrintCaptionID) (it cannot appear in both). This data is appended to the caption data.

'pnot' resource related data (keywords and caption) (Mac only)

'ANPA' resource (Mac only)

It is a bug that the TIFF information comes prior to the image resource information on this list. This means that an edit to the TIFF info will not be recognized unless the image resource information is removed. The TIFF data may be moved to after the image resource information in a future version of Photoshop.

Whenever writing a file and skipping bytes, write zeros.

Whenever reading one of the length delimited sections, use the length to decide whether you should stop reading.

When writing one of these sections, it is usually a good idea to write the entire section as Photoshop may endeavor to read the whole thing.

The areas not stored for a layer mask are set to 255.

The areas not stored for a transparency mask are fully transparent.

The following sections describe the information stored in the file, in order.

1. Signature (4 bytes)
Always equal to '8BPS' for this format. Do not try to read the file if the signature does not match this value.

2. Version (2 bytes)
Always equal to 1 for this format. Do not try to read the file if the version number does not match this value.

3. Reserved (6 bytes)
Readers should ignore these bytes, and writers should write zeros.

4. Channels (2 bytes)

The number of channels in the image, including any alpha channels. Supported range is 1 to 24.

5. Rows (4 bytes)
The height of the image in pixels. Supported range is 1 to 30000.

6. Columns (4 bytes)
The width of the image in pixels. Supported range is 1 to 30000.

7. Depth (2 bytes)
The number of bits per channel. Supported values are 1, 8 and 16.

8. Image Mode (2 bytes)

The color mode of the file. Supported values are: Bitmap = 0, Grayscale = 1, Indexed Color = 2, RGB Color = 3, CMYK Color = 4, Multichannel = 7, Duotone = 8, Lab Color = 9.

9. Color Data (4 bytes length + variable)
Contains the required data to define the color mode.

For indexed color images, the count will be equal to 768, and the mode data will contain the color table for the image, in non-interleaved order.

For duotone images, the mode data will contain the duotone specification, the format of which is not documented. Non-Photoshop readers can treat the duotone image as a grayscale image, and keep the duotone specification around as a black box for use when saving the file.

For all other modes, the byte count is zero.

10. Image Resources (4 bytes length + variable)
Successive blocks of data in the following format:

1. Resource Type (4 bytes)

2. Resource ID (2 bytes)

3. Resource Name (a Pascal format string padded to make the size even)

4. Resource Size (4 bytes)

5. Resource Data (Resource Size bytes plus padding to make the size even)

If the resource type is '**8BIM**' then:

If the resource ID is 2999, then this resources contains a Pascal-style string containing the name of the clipping path to use with this image when saving it as an EPS file.

If the resource ID is in the range 2000 - 2998 then the resource is a path resource. The name of the resource is the name given to the path when it was saved.

All points used in defining a path are stored as a pair of 32-bit components, vertical component first. The two components are fixed point numbers with 8 bits before the binary point and 24 bits after the binary point. We insist on leaving three guard bits in the points to eliminate most concerns over arithmetic overflow. Hence, the range for each component is 0xF0000000 to 0x0FFFFFFF representing a range of -16 to 16. We include the lower bound but not the upper bound. We use such a limited range because we express the points relative to the image size. The vertical component is given with respect to the image height and the horizontal component is given with respect to the image width. <0,0> represents the top-left corner of the image; <1,1> (<0x01000000,0x01000000>) represents the bottom-right.

A path resource consists of a sequence of 26 byte records as follows:

A. The first two bytes (bytes 0 and 1) of each record are a 16-bit value which indicates the kind of data contained in the rest of the record.

B. If the kind value is 0, 1, or 2, then this record is part of the description of a closed subpath within the compound path.

> **1.** If the kind value is 0, then bytes 2 and 3 of the record contain the length of the closed subpath. Such a record is then followed by records describing the knots of the subpath. This must be the first record in the subpath description.
>
> **2.** If the kind value is 1 or 2, then the remaining 24 bytes of the record represent three points in the above format giving the control point for the Bezier segment preceding the knot, the anchor point for the knot, and the control point for the Bezier segment leaving the knot in that order. If the kind value is 1, the control points are linked; i.e., editing one point edits the other one to preserve collinearity. Knots should only be marked as having linked controls if their control points are collinear with their anchor. If the kind value is 2, then this is a knot for which the control points are not linked.

C. If the kind value is 3, 4, or 5, then this record is part of the description of an open subpath within the compound path.

> **1.** If the kind value is 3, then this is a path length record just like kind value 0.
>
> **2.** If the kind value is 4, then this record contains the data for a knot with linked controls on the open subpath.
>
> **3.** If the kind value is 5, then this record contains the data for a knot with non-linked control on the open subpath.

D. Further kind values may be added in the future. Since Photoshop will ignore records for which it does not understand the kind value, this is a relatively easy format to extend.

11. Misc Size (4 byte length + variable)

1. Layers Size (4 byte length)

Rounded to a multiple of 2.

Successive blocks of data in the following format:

2. Layer Count (2 bytes)

If negative then the first alpha channel is the merged transparency channel and the actual layer count is the abolute value of the number.

3. for each layer:

1. Layer Top (4 bytes)

2. Layer Left (4 bytes)

3. Layer Bottom (4 bytes)

4. Layer Right (4 bytes)

The above describe the rectangle containing the contents of the layer.

5. Layer Channels (2 bytes)

The number of channels in the layer.

for each channel:

1. Channel ID (2 bytes)

 0 = red, 1 = green, etc.
-1 = transparency mask
-2 = user supplied layer mask

2. Channel Data Length (4 bytes)

The length in bytes of the data for the channel.
See **Channel Data** below.

6. Blend Mode Signature (4 bytes)

always equal to '8BIM'.

7. Blend Mode Key (4 bytes)

'norm' = normal
'dark' = darken
'lite' = lighten
'hue ' = hue
'sat ' = saturation
'colr' = color
'lum ' = luminosity
'mul ' = multiply
'scrn' = screen
'diss' = dissolve
'over' = overlay
'hLit' = hard light
'sLit' = soft light
'diff' = difference

8. Opacity (1 byte)

0 = transparent .. 255 = opaque.

9. Clipping (1 byte)

0 = base,
1 = non-base.
In the future this may be extended to allow deeper nesting.

10. Flags (1 byte)

bit 0: transparency protected
bit 1: visible
bit 2: obsolete

11. Zero (1 byte)

12. Extra Data Size (4 bytes)

Extra Data Size bytes of data as follows:

1. Layer Mask Data Size (4 bytes)

Layer Mask Data Size bytes as follows:

A. Top (4 bytes)

B. Left (4 bytes)

C. Bottom (4 bytes)

D. Right (4 bytes)

E. Default Color (1 byte)

0 or 255.

F. Flags (1 byte)

bit 0: position relative to layer
bit 1: layer mask disabled
bit 2: invert layer mask when blending

G. Pad (2 bytes)

Zeros.

2. Layer Blending Ranges Length (4 bytes)

Layer Blending Ranges Length bytes as follows:

A. Grayscale Source Range (4 bytes)

Present but irrelevant for Lab & Grayscale.

Contains 2 black values followed by 2 white values.

B. Grayscale Destination Range (4 bytes)

C. First Channel Source Range (4 bytes)

D. First Channel Destination Range (4 bytes)

above repeated for remaining channels.

3. Layer Name (1 byte + variable)

Pascal style string containing the layer name and sufficient zeros to pad to a multiple of 4.

4. for each layer:

A. Channel Data (variable)

for each channel in the layer :

1. Compression (2 bytes)

0 = Raw Data,
1 = RLE compressed.

2. Image Data (variable)

If the compression code is 0, the image data is just the raw image data calculated as ((**Layer Bottom - Layer Top**) * (**Layer Right - Layer Left**)).

If the compression code is 1, the image data starts with the byte counts for all the scan lines in the channel (**Layer Bottom - Layer Top**), with each count stored as a two-byte value. The RLE compressed data follows, with each scan line compressed separately. The RLE compression is the same compression algorithm used by the Macintosh ROM routine PackBits, and the TIFF standard.

At this point, if the data since the **Layers Size** is odd, a pad byte will be inserted.

5. Layer Mask Alpha Size (4 bytes)

The next Layer Mask Alpha Size bytes have the following structure.

1. Overlay Color Space (2 bytes)

2. Color Components (8 bytes)

4 * 2 byte color components

3. Opacity (2 bytes)

0 = transparent,
100 = opaque.

4. Kind (1 byte)

0 = Color Selected -- i.e. inverted,
1 = Color Protected,
128 = use value stored per layer. This value is preferred.
The others are for backward compatibility with beta versions.

5. Pad (1 byte)

Zero.

12. Compression (2 bytes)

0 = Raw Data,
1 = RLE compressed.

13. Image Data (variable)

Image data is stored in planar order, e.g. all the red data, all the green data, etc. Each plane is stored in scanline order, with no pad bytes.

If the compression code is 0, the image data is just the raw image data.

If the compression code is 1, the image data starts with the byte counts for all the scan lines (rows * channels), with each count stored as a two-byte value. The RLE compressed data follows, with each scan line compressed separately. The RLE compression is the same compression algorithm used by the Macintosh ROM routine PackBits, and the TIFF standard.

EPS

Photoshop 3.0 writes a high-resolution bounding box comment to the EPS file immediately following the traditional EPS bounding box comment. The comment begins with "%%HiResBoundingBox" and is followed by four numbers identical to those given for the bounding box except that they can have fractional components (i.e., a decimal point and digits after it). The traditional bounding box is written as the rounded version of the high resolution bounding box for compatibility.

Photoshop writes its image resources out to a block of data stored as follows:

%BeginPhotoshop: <length> <hex data>

<length> is the length of the image resource data.

<hex data> is the image resource data in hexadecimal.

Photoshop includes a comment in the EPS files it writes so that it is able to read them back in again. Third party programs that write pixel-based EPS files may want to include this comment in their EPS files, so Photoshop can read their files.

The comment must follow immediately after the %% comment block at the start of the file.

The comment is:

%ImageData: <columns> <rows> <depth> <mode> <pad channels> <blockÊsize> <binary/hex> "<data start>"

<columns> is the width of the image in pixels.

<rows> is the height of the image in pixels.

<depth> is the number of bits per channel. Must be 1 or 8.

<mode> is the image mode. 1 for bitmap and gray scale images (determined by depth), 2 for Lab images, 3 for RGB images, and 4 for CMYK images.

<pad channels> is the number of other channels stored in the file, which are ignored when reading. (Photoshop uses this to include a gray scale image that is printed on non-color PostScript printers).

<block size> is the number of bytes per row per channel. This will be equal to (<columns> * <depth> + 7) / 8 if the data is stored in line-interleave format (or if there is only one channel), or equal to 1 of the data is interleaved.

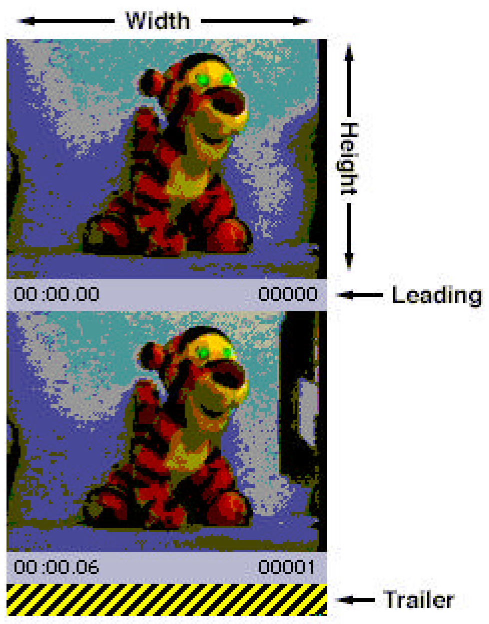
<binary/hex> is 1 if the data is in binary format, and 2 if the data is in hex format.

<data start> contains the entire PostScript line immediately preceding the image data. (This entire line should not occur elsewhere in the PostScript header code, but it may occur at part of a line.)

FilmStrip

Adobe Premiere 2.0 introduced a new file type: the Filmstrip. Premiere allows any video clip to be exported as a filmstrip. Adobe Photoshop 3.0 supports the filmstrip file type to allow each frame to be individually painted. The format for the filmstrip file is fairly simple, and is described below:

A Filmstrip consists of a sequence of equal sized 32-bit deep images, as shown in the picture below. The channel order in the file is Red, Green, Blue, Alpha. Between the frames is an arbitrarily sized leading area, in which any type of information may be embedded. Premiere puts the timecode and frame number for the frame in this area. This area is ignored by Premiere when the file is read, so the user is free to draw in this area. Following all the frames is a 16 row Trailer area the same width as the images. Premiere writes a yellow and black diagonal pattern in this area. The lower right corner of this area is actually an information record that exists at the very end of the file. This record is located by seeking to the end of the file minus the size of the record, then reading the record and verifying the signature field that it contains.



```
//------------------------------------------
// Definition for filmstrip info record

typedef struct {
  long    signature;  // 'Rand'
  long    numFrames;    // number of frames in file
  short   packing;    // packing method
  short   reserved;   // reserved, should be 0
  short   width;      // image width
  short   height; // image height
  short   leading;     // horiz gap between frames
  short   framesPerSec; // frame rate
  char    spare[16];  // some spare data.
} FilmStripRec, **FilmStripHand;
```

The fields are defined as follows:

- **signature**
  This field must be set to the code 'Rand' and is used to verify the validity of the record.

- **numFrames**
  This is the total number of frames in the file.

- **packing**
  This is the packing method used, currently only a value of 0 is defined, for no packing.

- **width**
  The width of each image, in pixels.

- **height**
  The height of each image, in pixels.

- **leading**
  The height of the leading areas, in pixels.

- **framesPerSec**
  The rate at which the frames should be played.

To locate the filmstrip info record:
Seek to the end of the file minus (sizeof(FilmStripRec)), then read in the FilmStrip record. Check the signature field for the code 'Rand' to test for validity.

To locate the data for a particular frame:

Seek to (frame * width * (height+leading) * 4), then read (width * height * 4) bytes. If the data is being placed into a GWorld, the channels must be re-arranged from Red-Green-Blue-Alpha to Alpha-Red-Green-Blue.

To write a FilmStrip file:
Write each frame sequentially into the file, including the leading areas. Then write a block of ((width * (height+leading) * 4) - sizeof(FilmStripRec)) bytes. Then fill in and write the FilmStrip record to the file.

Note: The packing field should currently be zero. In the future packing methods may be defined for filmstrips, so any software which reads filmstrips should examine this field before opening the file.

TIFF

The same "Image Resources" information is stored in TIFF files under tag number 34377 as is stored in Photoshop 3.0 files (see Image Resource Block and Image Resources in the preceding sections).

The following table describes the standard TIFF tags and tag values that Photoshop 3.0 is able to read and write.

Tag	**Reads**	**Writes**
IFD	First IFD in file	Only one IFD per file
NewSubFileType	Ignored	0
ImageWidth	1 to 30000	1 to 30000
ImageLength	1 to 30000	1 to 30000
BitsPerSample	1, 2, 4, 8, 16 (all same)	1, 8, 16
Compression	1, 2, 5, 32773	1, 5
PhotometricInterpretation	0, 1, 2, 3, 5, 8	0 (1-bit), 1 (8-bit), 2, 3,5,8
FillOrder	1	No
ImageDescription	Printing Caption	Printing Caption
StripOffsets	Yes	Yes
SamplesPerPixel	1 to 16	1 to 16
RowsPerStrip	Any	Single strip if not compressed, multiple strips if compressed.
StripByteCounts	Required if compressed	Yes
XResolution	Yes	Yes
YResolution	Ignored (square pixels assumed)	Yes
PlanarConfiguration	1 or 2	1
ResolutionUnit	2 or 3	2
Predictor	1 or 2	1 or 2
ColorMap	Yes	Yes
TileWidth	Yes	No
TileLength	Yes	No
TileOffsets	Yes	No
TileByteCounts	Required if compressed	No
InkSet	1	No
DotRange	Yes, if CMYK	Yes
ExtraSamples	Ignored (except for count)	0

Load File Formats

Introduction

Many of Photoshop's image processing operations are controlled by dialogs that allow the saving of dialog settings into a file. These files can be loaded into the dialog at a later time, even for use in a different image. Each load file has a unique file type and file extension associated with it. Photoshop for Macintosh will recognize either, but does not require the use of the extension. Photoshop for Windows will look for the given file extension automatically; this can be overridden.

Many, but not all, of the files have version numbers written as short integers in the first two bytes of the file. On the Macintosh, there is no information stored in the resource forks of any of Photoshop's load files. The files are completely interchangable with Photoshop for Windows or any other platform. Note that this requires consistent byte ordering between the all platforms when reading and writing these files. Photoshop stores multi-byte values with the high-order bytes first (big-endian), i.e. the reverse of the standard way this is done on Intel platforms (little-endian).

Arbitrary Map

Arbitrary Map files are loaded and saved in Photoshop's Curves dialog.

The Macintosh file type code is '**8BLT**'. The Windows file name extension is "**.AMP**".

There is no version number written in the file.   The file must be an even multiple of 256 bytes long.
Each 256 bytes is a lookup table, where the first byte corresponds to zero in the image data and the last byte to 255 in the image data. A "null" table that has no effect on an image is a linear table of bytes from 0 to 255.
If there is one table in the file, Photoshop applies it to the master composite channel, if the image has one, or to the single active channel if there is only one. If there is no composite channel, but more than one active channel, the load operation will have no effect. If the file has exactly three tables then it is assumed to represent an RGB lookup table and they are applied to the first channels in the image (the master composite map is untouched). If there is a single active channel, then the RGB lookup table is converted to grayscale and the result is applied to the active channel. In any other case, the first map is treated as a master and the remainder are applied to the image channels in turn (i.e. the second map is associated with the first channel, the third map with the second channel, etc.)
Photoshop handles single active channels in a special fashion. When saving the map applied to a single channel, only one map is written to the file. Similarly, when reading a map file for application to a single active channel, the master map is the one that will be used on that channel. This allows easy application of a single file to both composite and Grayscale images.

Brushes

Brushes settings files are loaded and saved in Photoshop's Brushes palette.

The Macintosh file type code is '**8BBR**'. The Windows file name extension is "**.ABR**".

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Count (2 bytes)
A short integer indicating how many brushes are in the remainder of the file.

3. Brushes (variable)
Two types of brushes are currently supported: elliptical, computed brushes and sampled brushes. Computed brushes are created with the New Brush command; sampled brushes are created from selected image data using the Define Brush command.

Each brush contains the following components:

a. type (2 bytes)
A short integer indicating the type of brush. A value of 1 means a computed brush, a value of 2 means a sampled brush. Other values are currently undefined.

b. size (4 bytes)
A long integer indicating the number of bytes in the remainder of the brush definition. Photoshop uses this information to skip over brush types that it doesn't understand.

c. data (**size** bytes)
The contents depend on the type of brush. Computed brush data is always 14 bytes; sampled brush data varies in size depending on the image data that makes up the brush tip.

Computed brushes:

i. miscellaneous (4 bytes): a long value which is ignored.

ii. spacing (2 bytes): a short integer ranging from 0 to 999 (0 means no spacing)

iii. diameter (2 bytes): a short integer ranging from 1 to 999

iv. roundness (2 bytes): a short integer ranging from 0 to 100

v. angle (2 bytes): a short integer ranging from -180 to 180

vi. hardness (2 bytes): a short integer ranging from 0 to 100

Sampled brushes:

i. miscellaneous (4 bytes): a long value which is ignored.

ii. spacing (2 bytes): a short integer ranging from 0 to 999 (0 means no spacing)

iii. anti-aliasing (1 byte): indicates whether the brush is to be anti-aliased when applied; 0 means no anti-aliasing. (Note that brushes with sampled data size either taller or wider than 32 pixels will not be anti-aliased by Photoshop in any event.)

iv. bounds (8 bytes): a rectangle, four short integers giving the bounds of the sampled tip data (in the order top, left, bottom, right)

v. bounds2 (16 bytes): a rectangle, exactly repeating the previous **bounds** entry, but in four long integers instead of four short integers.

vi. depth (2 bytes): depth of the sampled data, which is always 8

vii. image data (variable): if the bounds are taller than 16384, the data is broken into 16384-line chunks. Each chunk is streamed as follows:

> **a. compression** (2 bytes): two values are currently defined: 0 = Raw Data, 1 = RLE compressed
>
> **b. data** (variable): the brush tip image data is a single plane of grayscale data, stored in scanline order, with no pad bytes.

If the compression code is 0, the data is just the raw image data.

If the compression code is 1, the data starts with the byte counts for all the scan lines (equal to the number of rows, as described by the **bounds**), with each count stored as a two-byte value. The RLE compressed data follows, with each scan line compressed separately. The RLE compression is the same compression algorithm used by the Macintosh ROM routine PackBits, and the TIFF standard.

Color Table

Color Table files are loaded and saved in Photoshop's Color Table dialog (used with Indexed Color images), and can be loaded into the Colors palette as well.

The Macintosh file type code is '**8BCT**'. The Windows file name extension is **".ACT"**.

There is no version number written in the file. The file is expected to be exactly 768 bytes long.

256 RGB colors are written one at a time, starting with the first color in the table (index 0), with three bytes per color, in the order red, green, blue.

If loaded into the Colors palette, the colors will be installed in the color swatch list as RGB colors.

Colors

Colors files are loaded and saved in Photoshop's Colors palette.

The Macintosh file type code is '**8BCO**' The Windows file name extension is **".ACO"**.

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Count (2 bytes)
A short integer indicating how many colors are in the remainder of the file.

3. Colors (Count * 10 bytes)
Each color is ten bytes in size, and is made up of the following subsections:

a. color space (2 bytes)
A short integer indicating the color space the color is in, referred to below as the space ID.

b. color data (8 bytes)
Four short integers (possibly unsigned) that are the actual color data. If the color does not require four values to specify, the extra values are undefined and should be written as zeroes. The most basic color spaces are outlined below.

RGB colors have a space ID of 0. The first three values in the color data are, respectively, the color's red, green, and blue components. They are full unsigned 16-bit values as in Apple's RGBColor data structure (e.g. pure red is defined as 65535, 0, 0).

HSB colors have a space ID of 1. The first three values in the color data are, respectively, the color's hue, saturation, and brightness components. They are full unsigned 16-bit values as in Apple's HSVColor data structure (e.g. pure red is defined as 0, 65535, 65535).

CMYK colors have a space ID of 2. The four values in the color data are, respectively, the color's cyan, magenta, yellow, and black components. They are full unsigned 16-bit values, with 0 representing 100% ink (e.g. pure cyan is defined as 0, 65535, 65535, 65535).

Lab colors have a space ID of 7. The first three values in the color data are, respectively, the color's lightness, a chrominance, and b chrominance components. The lightness component is a 16-bit value ranging from 0 to 10000.   The chromanance components are each 16-bit values ranging from -12800 to 12700. Gray values are represented by chrominance components of 0 (e.g. pure white is defined as 10000, 0, 0).

Grayscale colors have a space ID of 8. The first value in the color data is the gray value; it ranges from 0 to 10000.

Photoshop allows the specification of custom colors, such as those colors that are defined in a set of custom inks provided by a printing ink manufacturer. These colors can be stored in the Colors palette and streamed to and from load files. The details of a custom color's color data fields are not public and should be treated as a black box. However, the following list gives the color space IDs currently defined by Photoshop for some custom color spaces:

Custom Color Space	Space ID
FOCOLTONE COLOUR SYSTEM	4
HKS colors (European Photoshop only)	10
PANTONE MATCHING SYSTEM	3
TOYO 88 COLORFINDER 1050	6
TRUMATCH colors	5

Command Buttons

Commands settings files are loaded and saved in Photoshop 3.0's Commands palette. This feature supplants the Function Key feature of Photoshop 2.5. The Commands palette buttons are simple mappings to Photoshop menu items, with optional function key shortcut and colorization.

The Macintosh file type code is '**8BFK**'. The file name extension is '**.ACM**'.

1. Version (2 bytes)
Equal to 2, written as a short integer.

2. Count (2 bytes)
The number of command records that follow. There are no pad bytes between records.

3. Command Records (variable)
The remainder of the file contains the Command records, one after the other. Each one is composed of the following:

a. Command ID (4 bytes)
This field is obsolete and must be set to zero.

b. Function Key ID (2 bytes)
This is an integer ranging from -15 to 15. Positive numbers map directly onto the numbered function keys (F1, F2, etc.) that are present on many personal computer keyboards. Negative numbers indicate that the shift key must be used as well for the keyboard shortcut (Shift-F1, Shift-F2, etc.). Zero means the button has no keyboard shortcut. On Windows systems, values outside of -12 to 12 will be ignored as standard Windows systems have 12 function keys on the keyboard. Windows systems will also map 1 to 0, as the F1 key is reserved for calling up Help. These numnbers should be unique across all entries in a Commands file, however Photoshop will ignore duplicates.

c. Color Index (2 bytes)
Each command button can be assigned a color with which its background will be tinted when drawn. There are eight predefined colors, with matching values as follows: 0 = None (button drawn in black-and-white), 1 = Red, 2 = Orange, 3 = Yellow, 4 = Green, 5 = Blue, 6 = Purple, 7 = Gray.

d. Title Matching Flag (1 byte)
If set to 1, this boolean flag indicates that the button title should automatically be updated to match the command's current menu item text. For example, a button assigned to the Layers palette would change text from "show Layers" to "Hide Layers" automatically as the state of the palette and the actual menu item changes. If set to 0, the button title has been changed from the menu item text by the user and shouldn't change unless changed by the user again.

e. Button Title (variable)
This is the title of the button that will be drawn on the Command palette. It usually matches the corresponding menu item text. It is stored as a Pascal-style string, with no pad bytes.

f. Command Key (variable)
This is the key for finding the menu item in Photoshop's menus. To distinguish menu items from each other, which could be duplicated on different menus, a key may include the title of the menu itself followed by a colon (e.g. "Mode:RGB Color"). This text is displayed in the options dialog for the button, but not on the Commands palette itself. (Note that even if the Title Matching flag is turned on, the title of the button text on the screen never contains the menu title qualifier.) It is stored as a Pascal-style string, with no pad bytes.

Curves

Curves settings files are loaded and saved in Photoshop's Curves dialog and Black Generation curve dialog (from within Separation Setup Preferences). Curves files can also be loaded into any of Photoshop's transfer function dialogs, such as the Duotone Curve dialog from within Duotone Options. (When loaded into a transfer function dialog, only the first curve in a Curves file is used.)

The Macintosh file type code is '**8BSC**'. The Windows file name extension is **".CRV"**.

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Count (2 bytes)
A short integer indicating how many curves are in the file-- must be in the range 1 to 27.

3. Curves (variable)
The remainder of the file contains the curves, one after the other.

Each curve is written as follows (i.e. each curve is made up of the following subsections):

a. point count (2 bytes)
A short integer in the range 2 to 19 indicates how many points are in the curve.

b. curve points (point count * 2 bytes)
Each curve point is a pair of short integers where the first number is the output value (vertical coordinate on the Curves dialog graph) and the second is the input value. All coordinates have range 0 to 255.
Hence a null curve (no change to image data) is represented by the following five-number, ten-byte sequence in a file: 2 0 0 255 255 . (Note that Photoshop allows the option of displaying ink percentages instead of pixel values; this is a display option only and the internal data is unchanged, with 100% ink equal to image data of 0 and 0% ink equal to image data of 255.)
Generally, the first of the curves is a master curve that applies to all of the composite channels in a composite image mode (e.g. the Red, Green, and Blue channels are all modified by the master curve for an RGB document). The remaining curves apply to the active channels individually- the second curve applies to channel one (if it is an active channel), the third curve to channel two, etc., up until the seventeenth curve, which applies to channel sixteen. The exception to this, and the reason there are up to nineteen curves, is when the original image is indexed color. In this case three curves are created for the red, green, and blue portions of the image's color table, and they replace the curve that represents the first channel of the image. This adds two curves for indexed images, and so for indexed color images any alpha channel that is active corresponds to its channel number plus three (e.g. if channel two is active it corresponds to curve number 5).
Photoshop handles single active channels in a special fashion. When saving the curves applied to a single channel, the settings are stored into the master slot, at the beginning of the file. Similarly, when reading a curves file for application to a single active channel, the master curve is the one that will be used on that channel. This allows easy application of a single file to both RGB and Grayscale images.

Note that Photoshop 3.0 can write Curves files that Photoshop 2 will not be able to read, because Photoshop 3.0's active channel support is different from Photoshop 2.0's, and there could be more active channels in a

Curves dialog than 2.0 supported. Photoshop 3.0 will always write at least five curves to a curves file, for maximum compatability with version 2.0. However, beyond the curve for the fourth channel, it does not write null curves past the last non-null curve that has been specified in the dialog. The presence of extraneous null curves will not affect a load operation.
Also note that it is possible to create a Curves load file with Photoshop 3.0 that cannot be read by Photoshop 2.5; Photoshop 3.0 allows a maximum of 24 channels per document, Photoshop 2.5 allows 16. Such use of the Curves function is rare, however.

Duotone Options

Duotone settings files are loaded and saved in Photoshop's Duotone Options dialog.

The Macintosh file type code is '**8BDT**'. The Windows file name extension is **".ADO"**.

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Count (2 bytes)
A short integer indicating how many plates are in the duotone spec-- 1 for monotones, 2 for duotones, 3 for tritones, 4 for quadtones. Must be in the range 1 to 4.

3. Ink Colors (4 * 10 bytes)
Four ink colors, regardless of the number of plates. The contents of the colors beyond the last plate specified by **Count** are undefined. Each color is streamed in the same fashion as in the Colors load file, and consists of the following subsections:

> **a. color space** (2 bytes)
> A short integer indicating the color space the color is in.
>
> **b. color data** (8 bytes)
> Four short integers (possibly unsigned) that are the actual color data.

Please refer to the Colors file format for details on the contents of the color records.

4. Ink Names (4 * 64 bytes)
Four ink names, regardless of the number of plates. Each name is streamed as a Pascal-style string with a length byte followed by the characters in the string. Names may not be more than 63 characters in length. Each name is padded to occupy 64 bytes including the initial length byte. Any names beyond the last plate specified by **Count** should be the empty string (size = 0).

5. Ink Curves (4 * 28 bytes)
Four ink curves, regardless of the number of plates. Each curve has the following subsections:

> **a. transfer curve** (26 bytes)
> 13 short integers, each ranging from 0 to 1000 (representing 0.0 to 100.0). In addition, all but the first and last value may be -1 (representing no point on the curve). Hence a null transfer curve looks like this:
> 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1000.
>
> **b. override** (2 bytes)
> For compatability with Photoshop 2.0, this short integer should be 0. It is ignored by Photoshop 3.0.

Any curves beyond the last plate specified by **Count** should be equal to the null curve.

6. Dot Gain (2 bytes)
For compatability with Photoshop 2.0, this short integer should be 20. It is ignored by Photoshop 3.0.

7. Overprint Colors (11 * 10 bytes)
Eleven ink colors, regardless of the number of plates. The number of defined overprints depends on the number of plates, **Count**. For monotones, there are no overprint colors. For duotones, there is 1 overprint color. For tritones, there are 4 overprint colors. For quadtones, there are 11 overprint colors. The contents of the colors beyond the last defined overprint are undefined. Each color is streamed in the same fashion as in the Colors load file, and consists of the following subsections:

a. color space (2 bytes)
A short integer indicating the color space the color is in.

b. color data (8 bytes)
Four short integers (possibly unsigned) that are the actual color data.

Please refer to the Colors file format for details on the contents of the color records.

Halftone Screens

Halftone Screens settings files are loaded and saved in Photoshop's Halftone Screens dialog (from within Page Setup).

The Macintosh file type code is '**8BHS**'. The Windows file name extension is "**.AHS**".

1. Version (2 bytes)
Equal to 5, written as a short integer.

2. Screens (4 * 18 bytes)
Four screen descriptions, each of which has the following subsections:

a. frequency value (4 bytes)
This ink's screen frequency, in lines per inch. This is a binary fixed point value with sixteen bits representing each of the integer and fractional parts of the number. Values range from 1.0 to 999.999, with units in lpi (lines per inch).

b. frequency scale (2 bytes)
The units for the screen frequency. Line per inch = 1, lines per centimeter = 2. Does not affect the frequency value itself, merely the way the value will be displayed on the screen.

c. angle (4 bytes)
Angle for this screen, a binary fixed point value with sixteen bits representing each of the integer and fractional parts of the number. Values range from -180.0000 to 180.0000, measured in degrees.

d. shape code (2 bytes)
A code representing the shape of the halftone dots in this screen. Round = 0, Ellipse = 1, Line = 2, Square = 3, Cross = 4, Diamond = 6. Custom shapes are represented by a negative number. The absolute value of this number is the size in bytes of the custom Spot Function, which is outlined below.

e. miscellaneous (4 bytes)
For compatability, this should be set to 0. It is not currently used by Photoshop.

f. accurate screens (1 byte)
Boolean flag which is true (1) if accurate screens should be used, false (0) otherwise.

g. default screens (1 byte)
Boolean flag which is true (1) if printer's default screens should be used, false (0) otherwise.

3. Spot Functions (size is the sum of the absolute values of all negative shape codes)
For every screen which has a custom spot function, the text of the PostScript function is written here. The functions are written one after the other with no header information, in the same order as the screen settings (screen description 1's spot function, if it has one, followed by number 2's, etc.). The shape code for those screens that have custom functions provides enough information to separate the various functions and assign them.

Hue/Saturation

Hue/Saturation settings files are loaded and saved in Photoshop's Hue/Saturation dialog.

The Macintosh file type code is '**8BHA**'. The Windows file name extension is **".HSS"**.

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Mode (1 byte)
Photoshop's Hue/Saturation dialog has two overall modes: in one, the settings represent shifts in the image data's hue and saturation, in the other the entire image is colorized to a single hue. This byte is a boolean flag indicating whether the colorization data or the hue-adjustment data in the file should be used. If the byte is zero, the hue-adjustment data will be used. If the byte is non-zero (Photoshop writes it as a 1) the colorization data will be used. (Both sets of data are present, but only one is used depending on the value of this byte.)

3. Padding (1 byte)
This pad byte must be present but is ignored by Photoshop.

4. Colorization (6 bytes)
Three short integers representing colorization settings. All values are in the range -100 to 100. The first number is the hue in which the image data will be colorized; the user interface represents the range of values as -180 to 180, where the number represents the hue in the traditional HSB color wheel, with zero equal to red. The next number is the saturation, the third number is the lightness adjustment.

5. Hue-Saturation Settings (42 bytes)
This data consists of three sets of seven short integers; all values range from -100 to 100:

> **a. hue settings** (14 bytes)
> One master value and six other values. The first value is the master hue change. For RGB and CMYK images, the other six values apply to each of the six hextants in the HSB color wheel: those image pixels nearest to red, yellow, green, cyan, blue, or magenta. (These numbers appear in the user interface as being in the range -60 to 60; the values are nevertheless stored as -100 to 100 and the slider will reflect each of the possible 201 values.) For Lab images, the first four of these values are applied to image pixels in the four Lab color quadrants (yellow, green, blue, magenta), and the other two values are ignored (Photoshop sets them to zero). (The values that are used range from -90 to 90 in the user interface.)

b. saturation settings (14 bytes)
Seven short integers representing the saturation adjustments. The first is a master value. The other six are applied to pixels using the same hue sextant or quadrant breakdown as for the hue adjusments; as before the last two are ignored for Lab documents.

c. lightness settings (14 bytes)
The last seven short integers are the lightness adjustments. The first is a master value. The other six are applied to pixels using the same hue sextant or quadrant breakdown as for the hue and saturation adjusments; as before the last two are ignored for Lab documents.

Ink Colors Setup

Ink Colors settings files are loaded and saved in Photoshop's Ink Colors Setup dialog, via the Preferences submenu.

The Macintosh file type code is '**8BIC**'. The Windows file name extension is "**.API**".

1. Version (2 bytes)
Equal to 4, written as a short integer.

2. Ink Colors (27 * 2 bytes)
Nine short integer triples specifying the xyY (CIE) values for the inks and their combinations. The inks are specified in the order Cyan, Magenta, Yellow, Magenta-Yellow (Red), Cyan-Yellow (Green), Cyan-Magenta (Blue), Cyan-Magenta-Yellow, followed by the White and Black points. Each triple is written in the order x (range: 0 to 10000, representing 0.0 to 1.0000), y (range: 1 to 10000, representing 0.0001 to 1.0000), Y (range: o to 20000, representing 0.00 to 200.00).

3. Gray Balance (4 * 2 bytes)
Four short integers specifying the gray color balance for Cyan, Magenta, Yellow, and Black. Each ranges from 50 to 200 (representing 0.5 to 2.00).

4. Dot Gain (2 bytes)
A short integers specifying the dot gain. Ranges from -10 to 40 (-10% to 40%).

Custom Kernel

Kernel settings files are loaded and saved in Photoshop's Custom filter dialog.

The Macintosh file type code is '**8BCK**'. The Windows file name extension is "**.ACF"**.

There is no version number written in the file. The file is expected to be exactly 54 bytes long, representing 27 short integers.

1. Weights (50 bytes)
The first 25 values are the custom weights, applied to pixels offset from (-2, -2) to (2, 2) off of each image pixel. The values progress through horizontal offsets first, e.g. the first five values all represent a vertical offset of -2. Each value can range from -999 to 999.

2. Scale (2 bytes)
This value can range from 1 to 9999.

3. Offset (2 bytes)
This value can range from -9999 to 9999.

Levels

Levels settings files are loaded and saved in Photoshop's Levels dialog.

The Macintosh file type code is '**8BLS**'. The Windows file name extension is "**.ALV**".

There are two versions of this file format. Photoshop 3.0 reads both but only writes version 2. Note that because the maximum number of channels that a document can contain was increased in Photoshop 3.0 (from 16 to 24), Photoshop 3.0 actually writes a longer Levels file than Photoshop 2.5. Photoshop 2.5 is still capable of reading these files, however, and will simply ignore the extra data.

1. Version (2 bytes)
Equal to 2, written as a short integer.

2. Levels Records (290 bytes)
Twenty-nine sets of levels. Each set of levels consists of five short integers, in ten bytes. The first number in a set is the input floor setting, and must range from 0 to 253. The second number is the input ceiling, and must range from 2 to 255. Third is the output value to which the input floor willbe matched. It can range from 0 to 255. Fourth is the ceiling output, also ranging from 0 to 255. The fifth value is the gamma to be applied to the image data. It ranges from 10 to 999 (representing the values 0.1 to 9.99).
The first set of levels are the master levels that apply to all of the composite channels in a composite image mode (e.g. the Red, Green, and Blue channels are all modified by the master levels settings for an RGB document). The remaining sets apply to the active channels individually--the second set applies to channel one (if it is an active channel), the third set to channel two, etc., up until the 25th set, which applies to channel 24. The exception to this is when the original image is indexed color. In this case three sets of levels are created for the red, green, and blue portions of the image's color table, and they replace the levels that represent the first channel of the image. This adds two sets of levels for indexed images, and so for indexed color images any alpha channel that is active corresponds to its channel number plus three (e.g. if channel two is active it corresponds to set number 5). The 28th and 29th sets are reserved for future use and should be set to zeroes.
Photoshop handles single active channels in a special fashion. When saving the levels applied to a single channel, the settings are stored into the master slot, at the beginning of the file. Similarly, when reading a levels file for application to a single active channel, the master levels are the ones that will be used on that channel. This allows easy application of a single file to both RGB and Grayscale images.

Monitor Setup

Monitor settings files are loaded and saved in Photoshop's Monitor Setup dialog, via the Preferences submenu.

The Macintosh file type code is '**8BMS**'. The Windows file name extension is "**.AMS**".

1. Version (2 bytes)
Equal to 2, written as a short integer.

2. Gamma (2 bytes)
A short integer indicating the monitor's gamma. Must be in the range 75 to 300 (representing 0.75 to 3.00).

3. White Point (2 * 2 bytes)
Two short integers giving the monitor's white point: the first is the x value, ranging from 0 to 10000 (representing 0.0 to 1.0000), the second is the y value, ranging from 1 to 10000 (representing 0.0001 to 1.0000).

4. Phosphors (6 * 2 bytes)
Three sets of two integers giving the x-y coordinates of the red, green, and blue phosphors. First comes red x, then red y; then green x, etc. The x values range from 0 to 10000 (representing 0.0 to 1.0000); the y values range from 1 to 10000 (representing 0.0001 to 1.0000).

Replace Color/Color Range

Replace Color settings files are loaded and saved in Photoshop's Replace Color dialog. They are also used to load and save settings from the Color Range dialog

The Macintosh file type code is **'8BXT**'. The file name extension is '**.AXT'**.

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Color Space (2 bytes)
A short integer indicatin what space the color components are in. 7
indicates Lab color, 8 indicates Grayscale. No other values are supported.

3. Component Ranges (6 bytes)
These six unsigned byte values represent the range of colors within which a pixel's color must fall to be considered selected for color replacement, or color range selecting. If the Color Space is grayscale, the first two bytes are the low and high endpoints of the range of gray values that are to be selected. The other four bytes should be zeroed. If the Color Space is Lab, then the first two bytes are the low and high endpoints of a range of 'L' values, the second two bytes are the low and high endpoints of a range of 'a' chromanance values, and the third pair bytes are the low and high endpoints of a range of 'b' chromanance values.

4. Fuzziness (2 bytes)
This short integer records the fuzziness setting, which controls how colors close to the selected colors are to be affected. It ranges from 0 to 200.

5. Transform Settings (6 bytes)
For files loaded into the Color Range dialog, these values are ignored. The Color Range dialog will write zeroes here. For Replace Color, this consists of three short integers; all values range from -100 to 100:

a. hue transform (2 bytes)
The hue change to be applied to the selected colors.

b. saturation transform (2 bytes)
The saturation change to be applied to the selected colors.

c. lightness transform (2 bytes)
The lightness change to be applied to the selected colors.

Scratch Area

Scratch Area settings files are loaded and saved in Photoshop's Scratch palette.

The Macintosh file type code is '**8BSR**'. The file name extension is '**.ASR**'.

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Scratch Area data (variable)
The Photoshop scratch area consists of RGB image data. The three planes of data are written one after the other, in the order Red, Green, Blue; each consists of the following:

> **a. bounds** (16 bytes): a rectangle, four long integers giving the bounds of the scratch data (in the order top, left, bottom, right); for Photoshop 3.0, this must always correspond to [0, 0, 89, 200] as the Scratch palette has a fixed size.
>
> **b. depth** (2 bytes): depth of the current plane of data, which is always 8.
>
> **c. image data** (variable):
>
> > **i. compression** (2 bytes): two values are currently defined: 0 = Raw Data,
> > 1 = RLE compressed
> >
> > **ii. data** (variable): each plane of the scratch image data is stored in
> > scanline order, with no pad bytes.
> >
> > If the compression code is 0, the data is just the raw image data.
> >
> > If the compression code is 1, the data starts with the byte counts for all the scan lines (equal to the number of rows, as described by the bounds), with each count stored as a two-byte value. The RLE compressed data follows, with each scan line compressed separately. The RLE compression is the same compression algorithm used by the Macintosh ROM routine PackBits, and the TIFF standard.

Selective Color

Selective Color settings files are loaded and saved in Photoshop's Selective Color dialog.

The Macintosh file type code is '**8BSV**'. The file name extension is '**.ASV**'.

1. Version (2 bytes)
Equal to 1, written as a short integer.

2. Correction Method (2 bytes)
A short integer indicating how the color correction is to be applied: in Relative (0) or Absolute (1) mode.

3. Plate Corrections (80 bytes)
The remainder of the file contains 10 correction records, one after the other.

Each record is written as follows:

a. cyan correction (2 bytes)
A short integer in the range -100 to 100 indicating the amount of correction for the cyan component.

b. magenta correction (2 bytes)
A short integer in the range -100 to 100 indicating the amount of correction for the magenta component.

c. yellow correction (2 bytes)
A short integer in the range -100 to 100 indicating the amount of correction for the yellow component.

d. black correction (2 bytes)
A short integer in the range -100 to 100 indicating the amount of correction for the black component.

The first record is ignored by Photoshop 3.0 and is reserved for future use. It should be set to all zeroes. The rest of the records apply to specific areas of colors or lightness values in the image, in the following order: Reds, Yellows, Greens, Cyans, Blues, Magentas, Whites, Neutrals, Blacks.

Separation Setup

Separation settings files are loaded and saved in Photoshop's Separation Setup dialog, via the Preferences submenu.

The Macintosh file type code is '**8BSS**'. The Windows file name extension is "**.ASP**".

1. Version (2 bytes)
Equal to 300, written as a short integer.

2. Separation Type (1 byte)
A boolean flag indicating UCR (value = 0) or GCR (value = 1) separations.

3. Black Limit (2 bytes)
A short integer giving the black ink limit, ranging from 0 to 100.

4. Total Limit (2 bytes)
A short integer giving the total ink limit, ranging from 200 to 400.

5. UCA Amount (2 bytes)
A short integer giving the undercolor addition for GCR separations, ranging from 0 to 100.

6. Black Generation Curve (variable)
This is a spline curve. The format is identical to a single curve instance from the Curves file format. It is composed of two parts:

> **a. point count** (2 bytes)
> A short integer in the range 2 to 19 indicates how many points are in the curve.
>
> **b. curve points** (point count * 2 bytes)
> Each curve point is a pair of short integers where the first number is the output value (vertical coordinate on the Black Generation dialog graph) and the second is the input value. All coordinates have range 0 to 255.

Hence a null curve (no change to input values) is represented by the following five-number, ten-byte sequence in a file: 2 0 0 255 255 .

Note that the black generation curve and the UCA limit must both be present even if the Separation Type is set to UCR.

Separation Tables

Separation Table files are loaded and saved in Photoshop's Separation Tables dialog.

The Macintosh file type code is '**8BST**'. The Windows file name extension is **".AST"**.

If the size of the file is 33 * 33 * 33 * 4, then the file consists only of an Lab->CMYK table as currently documented.

If the size of the file is 33 * 33 * 33 + 256 * 3, then the file consists only of a CMYK->Lab table as currently documented.

Otherwise, we expect the file to have the following format.

1. Version (2 bytes)
Equal to 300, written as a short integer.

2. Has Lab to CMYK (1 byte)
Boolean indicating whether the file contains an Lab to CMYK table.

3. Has CMYK to Lab (1 byte)
Boolean indicating whether the file contains an CMYK to Lab table.

4. Lab to CMYK Table (33 * 33 * 33 * 4 bytes, optional)
If field 2 is equal to 1 (true), this section contains the CMYK colors for 33 * 33 *33 Lab colors. The Lab colors that are the source colors for this can be generated:

```
for (i = 0; i< 33;  i++)
  for (j = 0; j < 33; j++)
      for (n = 0; n < 33; n++)
      {
          L = Min (i * 8, 255);
          a = Min (j * 8, 255);
          b = Min (n* 8, 255);
      }
```

The CMYK colors are written in interleaved order, one byte each ink, 0 = 100%, 255 = 0%.

5. CMYK to Lab Table ((33 * 33 * 33 + 256) * 3 bytes, optional)
If field 3 is equal to 1 (true), this section contains the Lab colors for 33 * 33 *33 + 256 CMYK colors. The CMYK colors that are the source colors for this can be generated:

```
for (i = 0; i< 33;  i++)
  for (j = 0; j < 33; j++)
      for (n = 0; n < 33; n++)
      {
```

```
                c = Min (i * 8, 255);
                m = Min (j * 8, 255);
                y = Min (n* 8, 255);
                k = 255;
            }

      for (i = 0; i < 256; i++)
      {
        c = 255;
        m = 255;
        y = 255;
        k = i;
      }
```

The Lab colors are written in interleaved order, one byte per component.

If after reading the above data, the file is not yet empty, then the file
contains the following data.

6. Has Gamut Table (1 byte, 1 = have table, 0 = don't have table)
If the flag indicates the table is present, then the table data consists of (((33 * 33 * 33L) + 7) >> 3) bytes of data. This is a bit table indexed in the same way as the Lab->CMYK table with the provision that the high bit of the first byte is at index 0, etc.

(i.e., to test the bit at bitIndex we use table [bitIndex >> 3] & (0x0080 >> (bitIndex & 0x07))) != 0. bitIndex itself is calculated in the same way one would calculate an index into the Lab->CMYK table)

A 1 indicates that the color is in gamut and a 0 indicates that it is out of gamut.

Transfer Function

Transfer Function settings files are loaded and saved in Photoshop's Duotone Curve dialog (from within Duotone Options) and Transfer Function dialogs (from within Page Setup). Transfer Function files can also be loaded into any of Photoshop's curves dialogs, such as the Curves color adjustment dialog.

The Macintosh file type code is '**8BTF**'. The Windows file name extension is **".ATF"**.

1. Version (2 bytes)
Equal to 4, written as a short integer.

2. Functions (112 bytes)
There are four transfer functions in the file. Each function is made up of the following subsections:

a. curve (26 bytes)
A transfer curve consists of 13 short integers, each ranging from 0 to 1000 (1000 represents the value 100.0). In addition, all but the first and last value may be -1 (representing no point on the curve). Hence a null transfer curve looks like this: 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1000.

b. override (2 bytes)
This is a boolean flag indicating whether the curve should override the printer's default transfer curve. If it is zero, the printer's curve will not be overridden.

Note again that the file always contains four functions. For example, when writing the printer transfer functions for Grayscale images Photoshop writes four copies of the single transfer function specified in the user interface.

QuickTime File Format Specification, May 1996

Developer Press

Contents

Figures and Tables

About the QuickTime File Format

This book describes the format and content of QuickTime files. It is intended for developers who need to work with QuickTime files outside the context of the QuickTime environment. For example, if you are developing a non-QuickTime application that imports QuickTime files, you need to understand the material in this book. On the other hand, if you are using QuickTime on any of its supported platforms, you do not necessarily need to be familiar with the file format information presented here.

This book describes QuickTime files in general, rather than how they are supported on a specific computing platform or in a specific programming language. As a result, the file format information is presented in a tabular manner, rather than in coded data structures. Similarly, field names are presented in English rather than as programming language tags. Furthermore, to the extent possible, data types are described generically. For example, this book uses "32-bit signed integer" rather than "`long`" to define a 32-bit integer value. Based on the information provided here, you should be able to create appropriate data structure specifications for your environment.

Finally, QuickTime files are used to store QuickTime movies, as well as other time-based data. If you are writing an application that parses QuickTime files, you should recognize that there may be non-movie data in the files.

For More Information

The *Apple Developer Catalog* (ADC) is Apple Computer's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple computer platforms. Customers receive the *Apple Developer Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. ADC offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *Apple Developer Catalog*, contact

Apple Developer Catalog
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	ORDER.ADC
Internet	order.adc@applelink.apple.com

QuickTime File Format

This document describes how QuickTime movies are stored on disk. The QuickTime file format is designed to accommodate the varied kinds of data that need to be stored in order to work with digital media. Because the file format can be used to describe almost any media structure, it is an ideal format for the exchange of digital media between applications, regardless of the platform on which the application may be running.

This document assumes that the reader is familiar with the basic concepts of digital video and digital audio, as well as with QuickTime. For a complete description of QuickTime concepts, including time coordinate systems and how spatial tracks are composited together, please refer to *Inside Macintosh: QuickTime*.

A QuickTime file stores the description of the media separately from the media data. The description, or meta-data, is called the **movie** and contains information such as the number of tracks, video compression format, and timing information. The movie also contains an index of where all the media data is stored. The media data is all of the actual sample data, such as video frames and audio samples. The media data may be stored in the same file as the QuickTime movie, in a separate file, or in several files.

Before explaining the specifics of how a QuickTime movie is stored, it is important to first understand the basic units that are used to construct QuickTime files. QuickTime uses two basic structures for storing information: **atoms** and **QT atoms**. Both atoms and QT atoms allow you to construct arbitrarily complex hierarchical data structures. Both also allow applications to ignore data they don't understand.

Atom types are specified by a four-character code. Apple Computer reserves all four-character codes consisting entirely of lower case letters.

Unless otherwise stated, all data in a QuickTime movie is stored in big-endian (Motorola) byte ordering.

Finally, all version fields must be set to 0, unless this document states otherwise.

Atoms

The basic data unit in a QuickTime file is the atom**.** Each atom contains size and type information along with its data. The `size` field indicates the number of bytes in the atom, including the `size` and `type` fields. The `type` field specifies the type of data stored in the atom and, by implication, the format of that data. Both the `size` and `type` fields are 32-bit integers.

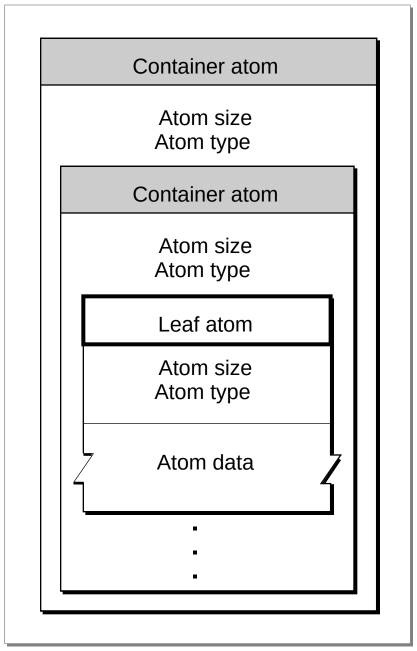
Atoms are recursive in nature. That is, one atom may contain one or more other atoms of varying type. For example, a movie atom contains one track atom for each track in the movie. The track atoms, in turn, contain one or more media atoms each, along with other atoms that define other track and movie characteristics. This hierarchical structure of atoms is referred to as a containment hierarchy.

The format of the data stored within a given atom cannot be determined based only on the `type` field of that atom. That is, an atom's use is determined by its context. A given atom type may have different usages when stored within atoms of different types. This means that all QuickTime file readers must take into consideration not only the atom type, but the atom's containment hierarchy.

Figure 1-1 shows the layout of a sample QuickTime atom. Each atom carries its own size and type information as well as its data. Throughout this chapter, the name of a **container atom** (an atom that contains other atoms, including other container atoms) is printed across a horizontal gray band, and the name of a **leaf atom** (an atom that contains no other atoms) is printed across a horizontal drop shadow box. Leaf atoms contain data, usually in the form of tables.

Atoms within container atoms do not have to be in any particular order, with the exception of handler description atoms. Handler description atoms must come before their data. For example, a media handler description atom must come before a media information atom. A data handler description atom must come before a data information atom.

Figure 1-1 A sample QuickTime atom



Atoms consist of a header, followed by atom data. An atom header consists of the following fields.

Field descriptions

Atom size	A 32-bit integer that indicates the size of the atom, including both the atom header and the atom's contents. If the atom is a leaf atom, then this field contains the size of the single atom. The size of a container atom includes all of its contained atoms.
Type	A 32-bit integer that contains the type of the atom.

The only way to interpret the atom's data is by knowing the type of data that is stored in an atom of a particular type.

QT Atoms

Given the limitations of the structure of the simple atom, Apple has created a new, enhanced data structure called a QT atom. QT atoms provide a more general purpose storage format and remove some of the ambiguities that arise when using simple atoms.

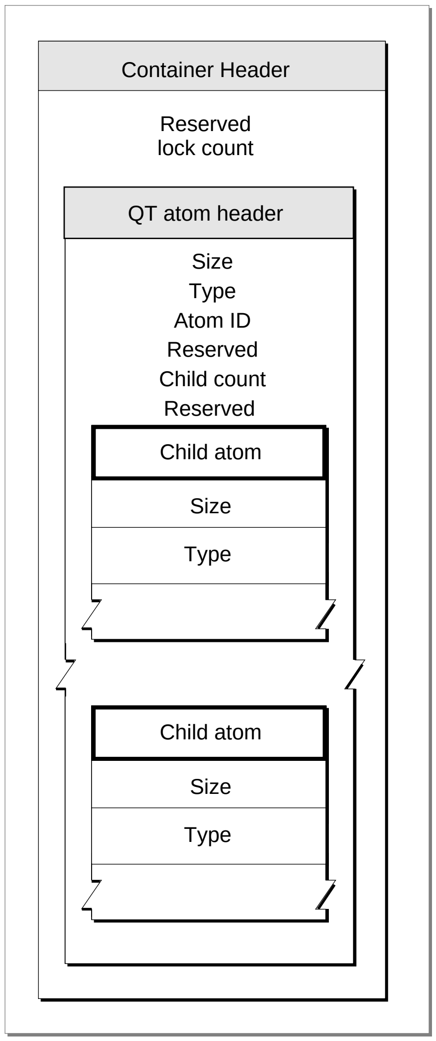
In particular, with simple atoms there is no way to know if an atom is a leaf node or whether it contains other atoms, or both, without specific knowledge about the atom. Using QT atoms, a given node is either is a leaf node or a container node. There is no ambiguity. Furthermore, QT atoms allow for multiple atoms of a given type to be specified through identification numbers. While QT atoms are a more powerful data structure, they require more overhead in the file.

The QuickTime file format uses both atoms and QT atoms. In general, newer parts of the QuickTime file format use QT atoms, and older parts use atoms. When defining new QuickTime structures, you should use QT atoms whenever practical.

Figure 1-2 depicts the layout of a QT atom. Each QT atom starts with a QT atom container header, followed by the root atom. The root atom's type is determined by the QT atom's type. The root atom contains any other atoms that are part of the structure.

Each container atom starts with a QT atom header followed by the atom's contents. The contents are either child atoms or data, but never both. If an atom contains children it also contains all of its children's data and their descendants. The root atom is always present and never has any siblings.

Figure 1-2 QT atom layout



A QT atom container header contains the following data.

Field descriptions

Reserved	A 10-byte element that must be set to 0.
Lock count	A 16-bit integer that must be set to 0.

Each QT atom header contains the following data.

Field descriptions

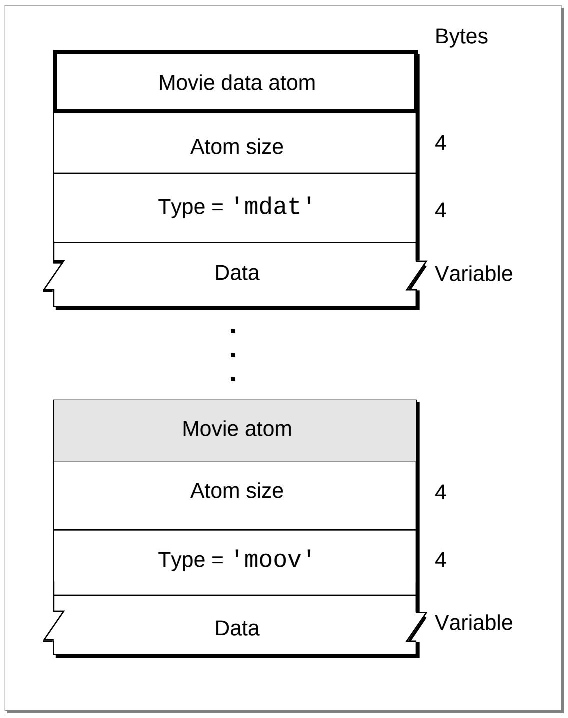
Size	A 32-bit integer that indicates the size of the atom in bytes, including both the QT atom header and the atom's contents. If the atom is a leaf atom, then this field contains the size of the single atom. The size of container atoms includes all of the contained atoms. You can walk the atom tree using the Size and Child count fields (described below).
Type	A 32-bit integer that contains the type of the atom. If this is the root atom, the type value is set to `'sean'`.
Atom ID	A 32-bit integer that contains the atom's ID value. This value must be unique among its siblings. The root atom always has an Atom ID value of 1.
Reserved	A 16-bit integer that must be set to 0.
Child count	A 16-bit integer that specifies the number of child atoms that an atom contains. This count only includes immediate children. If this field is set to 0, the atom is a leaf atom and only contains data.
Reserved	A 32-bit integer that must be set to 0.

File Format

A QuickTime file is simply a collection of atoms. QuickTime does not impose any rules about the order of these atoms.

Figure 1-3 depicts a typical QuickTime file.

Figure 1-3 The structure of a QuickTime file



In file systems that support file name extensions, QuickTime file names typically have an extension of `".mov"`. On the Macintosh platform, QuickTime files have a file type of `"MooV"`. On the Macintosh, the movie atom may be stored as a Macintosh resource using the Resource Manager. The resource has a type of `'moov'`. All media data is stored in the data fork.

As previously discussed, QuickTime files consist of atoms, each with an appropriate atom type. A few of these types are considered basic atom types and form the structure within which the other atoms are stored. Table 1-1 lists the currently supported basic atom types.

Table 1-1 QuickTime file basic atom types

Atom type	Use
`'free'`	Unused space available in file
`'skip'`	Unused space available in file
`'mdat'`	Movie data—usually this data can only be interpreted by using the movie resource
`'pnot'`	Reference to movie preview data
`'moov'`	Movie resource

While it is true that QuickTime imposes no strict order on a movie's atoms, it is often convenient if the movie atom appears near the front of the file. For example, an application that plays a movie over a network would not necessarily have access to the entire movie at all times. If the movie atom is stored at the beginning of the file, the application can use the meta-data to understand the movie's content as it is acquired over the network.

The following sections describe each of these basic atom types in more detail, including descriptions of the atoms that each basic atom may contain.

Free Space Atoms

Both free and skip atoms designate unused space in the movie data file. These atoms consist only of an atom header (atom size and type fields), followed by the appropriate number of bytes of free space. When reading a QuickTime movie, your application may safely skip these atoms. When writing or updating a movie, you may re-use the space associated with these atom types.

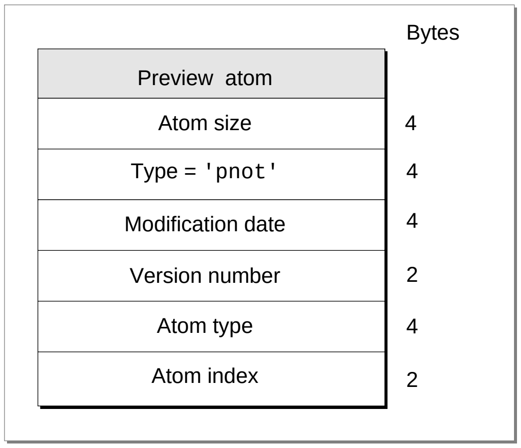
Movie Data Atoms

As with the free and skip atoms, the movie data atom is structured quite simply. It consists of an atom header (atom size and type fields), followed by the movie's media data. Your application can understand the data in this atom only by using the meta-data stored in the movie atom.

Preview Atoms

The preview atom contains information that allows you to find the preview image associated with a QuickTime movie. The preview image, or poster, is a representative image suitable for display to the user in, say, file-open dialogs. Figure 1-4 depicts the layout of a preview atom.

Figure 1-4 The layout of a preview atom



The preview atom has an atom type value of `'pnot'` and, following its atom header, contains the following fields.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this preview atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'pnot'`.
Modification date	A 32-bit unsigned integer containing a date that indicates when the preview was last updated. The date is in standard Macintosh format.
Version number	A 16-bit integer that must be set to 0.
Atom type	A 32-bit integer that indicates the type of atom that contains the preview data. Typically, this is set to `'PICT'` to indicate a QuickDraw picture.
Atom index	A 16-bit integer that identifies which atom of the specified type is to be used as the preview. Typically, this field is set to 1 to indicate that you should use the first atom of the type specified in the Atom type field.

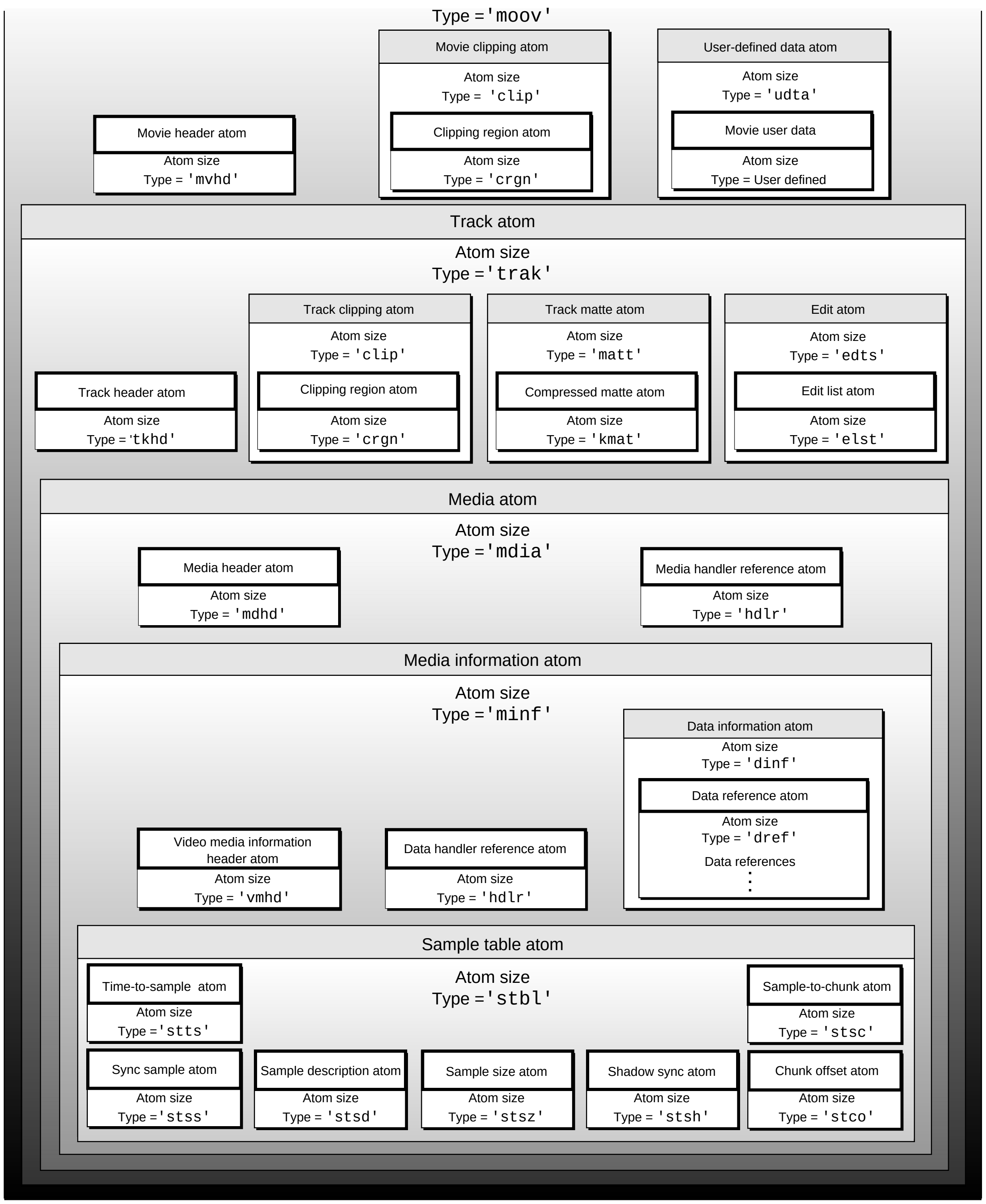
Movie Atoms

Movie atoms have an atom type of `'moov'`. These atoms act as a container for the information that describes a movie's data. This information, or meta-data, is stored in a number of different types of atoms. As such, the movie atom is essentially a container of other atoms. At the highest level, movie atoms contain track atoms, which in turn contain media atoms. At the lowest level you find the leaf atoms, which contain the actual data, usually in the form of a table or a data stream.

For example, the track atom contains the edit atom, which contains a leaf atom called the edit list atom. The edit list atom contains an edit list table. Both of these atoms are discussed later in this book.

Figure 1-5 provides a conceptual view of the organization of a simple, one-track QuickTime movie. Each nested box in the illustration represents an atom that belongs to its containing atom. The figure does not show the data regions of any of the atoms. These areas are described in the sections that follow.

Figure 1-5 Sample organization of a one-track video movie atom



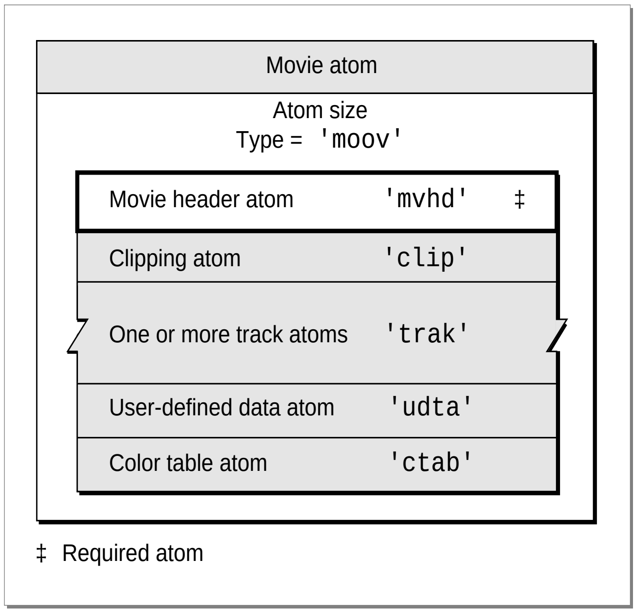
The Movie Atom

You use movie atoms to specify the information that defines a movie—that is, the information that allows your application to understand the data that is stored in the movie data atom. The movie atom contains the movie header atom, which defines the time scale and duration information for the entire movie, as well as its display characteristics. In addition, the movie atom contains each track in the movie.

The movie atom has an atom type of `'moov'`. It contains other types of atoms, including one leaf atom—the movie header (`'mvhd'`)—and several atoms that contain other atoms: a clipping atom (`'clip'`), one or more track atoms (`'trak'`), a color table atom (`'ctab'`), and user data (`'udta'`).

Figure 1-6 shows the layout of a movie atom. The movie header atom is the only required atom in the movie atom.

Figure 1-6 The layout of a movie atom



A movie atom contains the following information.

Field descriptions

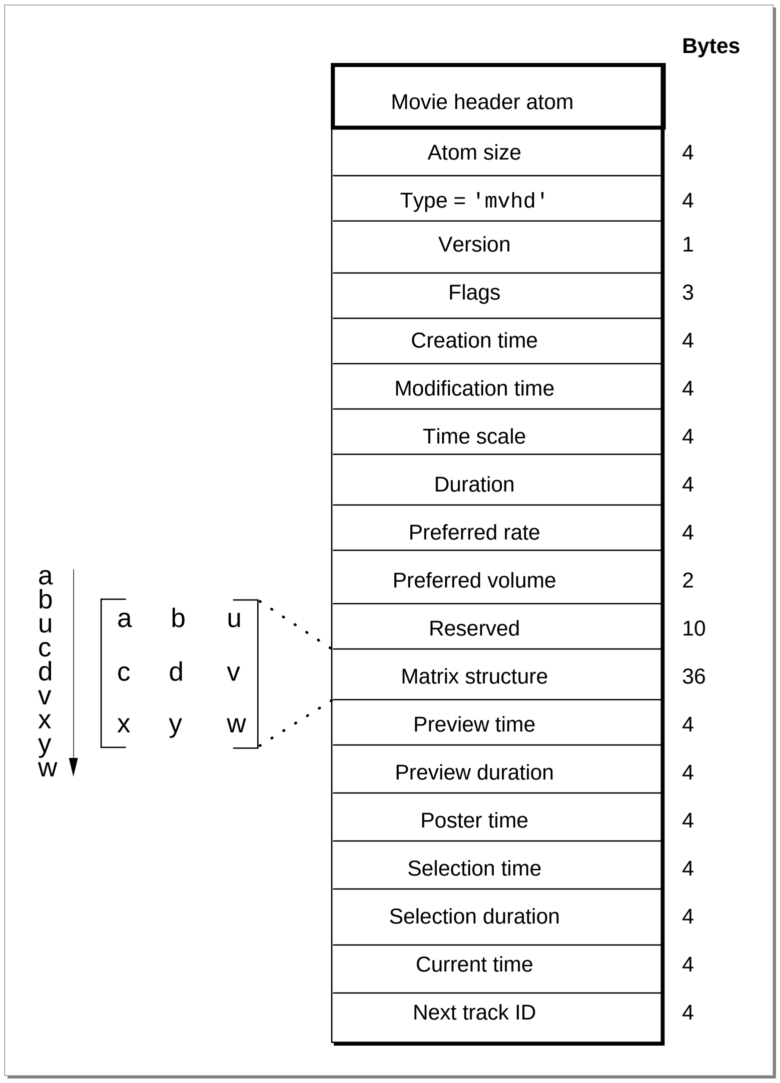
Size	The number of bytes in this movie atom.
Type	The type of this movie atom; this field must be set to `'moov'`.
Movie header	The movie header atom associated with this movie. See the next section for details on the movie header atom.
Movie clipping atom	The clipping atom associated with this movie. See “Clipping Atoms,” beginning on page 19, for more information.

Track list	One or more track atoms associated with this movie. See “Track Atoms,” beginning on page 15, for details on track atoms and their associated atoms.
User data	The user data atom associated with this movie. See “User Data Atoms” on page 57 for more information about user data atoms.
Color table	The color table atom associated with this movie. See “Color Table Atoms” on page 14 for a discussion of the color table atom.

Movie Header Atoms

You use the movie header atom to specify the characteristics of an entire QuickTime movie. The data contained in this atom defines characteristics of the entire QuickTime movie, such as time scale and duration. It has an atom type value of `'mvhd'`. Figure 1-7 shows the layout of a movie header atom. The movie header atom is a leaf atom.

Figure 1-7 The layout of a movie header atom



You define a movie header atom by specifying the following data elements.

Field descriptions

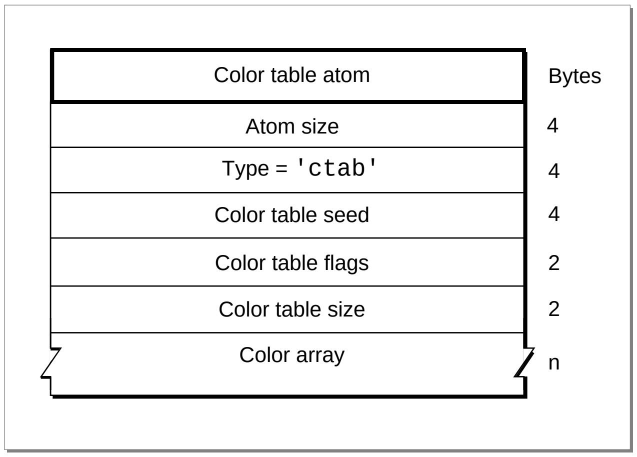
Size	A 32-bit integer that specifies the number of bytes in this movie header atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'mvhd'`.
Version	A 1-byte specification of the version of this movie header atom.
Flags	Three bytes of space for future movie header flags.
Creation time	A 32-bit integer that specifies (in seconds since midnight, January 1, 1904) when the movie atom was created.

Modification time	A 32-bit integer that specifies (in seconds since midnight, January 1, 1904) when the movie atom was changed.
Time scale	A time value that indicates the **time scale** for this movie—that is, the number of time units that pass per second in its time coordinate system. A time coordinate system that measures time in sixtieths of a second, for example, has a time scale of 60.
Duration	A time value that indicates the duration of the movie in time scale units. Note that this property is derived from the movie's tracks. The value of this field corresponds to the duration of the longest track in the movie.
Preferred rate	A 32-bit fixed-point number that specifies the rate at which to play this movie. A value of 1.0 indicates normal rate.
Preferred volume	A 16-bit fixed-point number that specifies how loud to play this movie's sound. A value of 1.0 indicates full volume.
Reserved	Ten bytes reserved for use by Apple. Set to 0.
Matrix	The matrix structure associated with this movie. A matrix shows how to map points from one coordinate space into another.
Preview time	The time value in the movie at which the preview begins.
Preview duration	The duration of the movie preview in movie time scale units.
Poster time	The time value of the time of the movie poster.
Selection time	The time value for the start time of the current selection.
Selection duration	The duration of the current selection in movie time scale units.
Current time	The time value for current time position within the movie.
Next track ID	A 32-bit integer that indicates a value to use for the track ID number of the next track added to this movie. Note that 0 is not a valid track ID value.

Color Table Atoms

Color table atoms define a list of preferred colors for displaying the movie on devices that only support 256 colors. The list may contain up to 256 colors. These optional atoms have a type value of `'ctab'`. The color table atom contains a Macintosh Color Table data structure. Figure 1-8 shows the layout of a color table atom.

Figure 1-8 The layout of a color table atom



The color table atom contains the following data elements.

Field descriptions

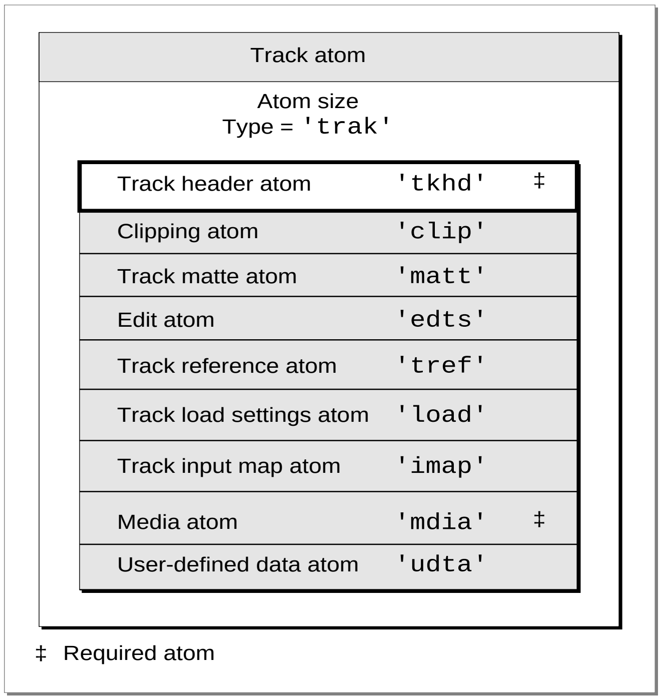
Size	A 32-bit integer that specifies the number of bytes in this color table atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'ctab'`.
Color table seed	A 32-bit integer that must be set to 0.
Color table flags	A 16-bit integer that must be set to 0x8000
Color table size	A 16-bit integer that indicates the number of colors in the following color array. This is a zero-relative value; setting this field to 0 means that there is one color in the array.
Color array	An array of colors. Each color is made up of four unsigned 16-bit integers. The first integer must be set to 0, the second is the red value, the third is the green value, and the fourth is the blue value.

Track Atoms

Track atoms define a single track of a movie. A movie may consist of one or more tracks. Each track is independent of the other tracks in the movie and carries its own temporal and spatial information. Each track atom contains its associated media atom.

Figure 1-9 shows the layout of a track atom. Track atoms have an atom type value of `'trak'`. The track atom requires the track header atom (`'tkhd'`) and the media atom (`'mdia'`). Other child atoms are optional and may include a track clipping atom (`'clip'`), a track matte atom (`'matt'`), an edit atom (`'edts'`), a track reference atom (`'tref'`), a track input map atom (`'imap'`), and a user data atom (`'udta'`).

Figure 1-9 The layout of a track atom


Track atom
Atom size
Type = 'trak'
Track header atom 'tkhd' ‡
Clipping atom 'clip'
Track matte atom 'matt'
Edit atom 'edts'
Track reference atom 'tref'
Track load settings atom 'load'
Track input map atom 'imap'
Media atom 'mdia' ‡
User-defined data atom 'udta'
‡ Required atom

Track atoms contain the following data elements.

Field descriptions

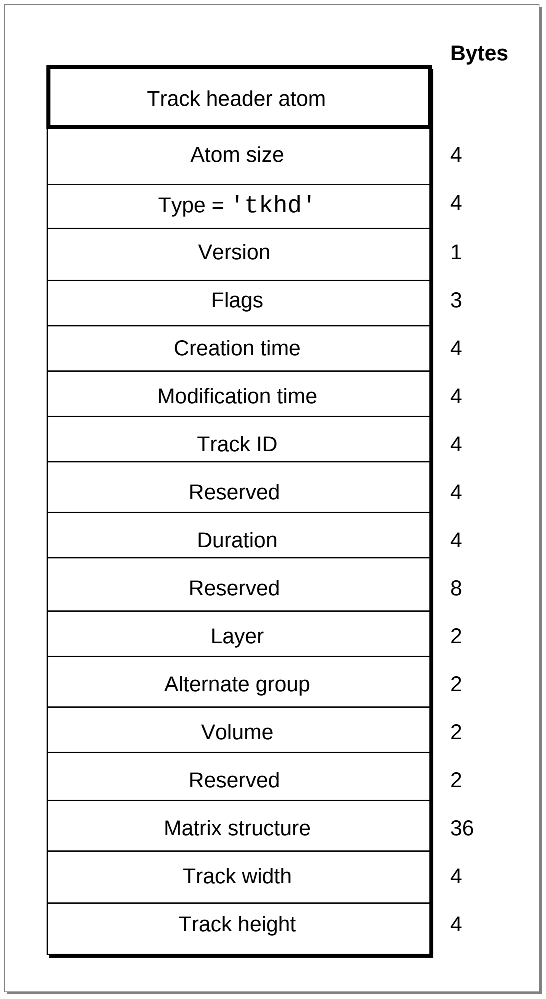
Size	A 32-bit integer that specifies the number of bytes in this track atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'trak'`.
Track header	The track header atom associated with this track. See the next section for details.
Track clipping	The track clipping atom associated with this track. See “Clipping Atoms,” beginning on page 19, for more information.
Track matte	The track matte atom associated with this track. See “Track Matte Atoms,” beginning on page 20, for more information.
Edits	The edit atom associated with this track. See “Edit Atoms,” beginning on page 22, for details.
Track references	The track reference atom associated with this track. See “Track Reference Atoms,” beginning on page 26, for details.
Track load settings	The track load settings atom associated with this track. See “Track Load Settings Atoms,” beginning on page 24, for details.
Track input map	The track input map atom associated with this track. See “Track Input Map Atoms,” beginning on page 27, for details.
Media	The media atom associated with this track. See “Media Atoms,” beginning on page 30, for details.

User data	The user data atom associated with this track. See “User Data Atoms” on page 57 for more information.

Track Header Atoms

The track header atom specifies the characteristics of a single track within a movie. A track header atom contains a Size field that specifies the number of bytes and a Type field that indicates the format of the data (defined by the atom type, `'tkhd'`). Figure 1-10 shows the structure of a track header atom.

Figure 1-10 The layout of a track header atom



The track header atom contains the track characteristics for the track, including temporal, spatial, and volume information.

Track header atoms contain the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this track header atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'tkhd'`.
Version	A 1-byte specification of the version of this track header.
Track header flags	Three bytes that are reserved for the track header flags. These flags indicate how the track is used in the movie. The following flags are valid (all flags are enabled when set to 1).

Track enabled	Indicates that the track is enabled. Flag value is 0x0001.
Track in movie	Indicates that the track is used in the movie. Flag value is 0x0002.
Track in preview	Indicates that the track is used in the movie’s preview. Flag value is 0x0004.
Track in poster	Indicates that the track is used in the movie’s poster. Flag value is 0x0008.

Creation time	A 32-bit integer that indicates (in seconds since midnight, January 1, 1904) when the track header was created.
Modification time	A 32-bit integer that indicates (in seconds since midnight, January 1, 1904) when the track header was changed.
Track ID	A 32-bit integer that uniquely identifies the track. A value of 0 must never be used for a track ID.
Reserved	A 32-bit integer that is reserved for use by Apple. Set this field to 0.
Duration	A time value that indicates the duration of this track (in the movie’s time coordinate system). Note that this property is derived from the track’s edits. The value of this field is equal to the sum of the durations of all of the track’s edits.
Reserved	An 8-byte value that is reserved for use by Apple. Set this field to 0.
Layer	A 16-bit integer that indicates this track’s spatial priority in its movie. The QuickTime Movie Toolbox uses this value to determine how tracks overlay one another. Tracks with lower layer values are displayed in front of tracks with higher layer values.
Alternate group	A 16-bit integer that specifies a collection of movie tracks that contain alternate data for one another. QuickTime chooses one track from the group to be used when the movie is played. The choice may be based on such considerations as playback quality or language and the capabilities of the computer.
Volume	A 16-bit fixed-point value that indicates how loudly this track’s sound is to be played. A value of 1.0 indicates normal volume.
Reserved	A 16-bit integer that is reserved for use by Apple. Set this field to 0.
Matrix	The matrix structure associated with this track. See Figure 1-44 on page 77 for an illustration of a matrix structure.

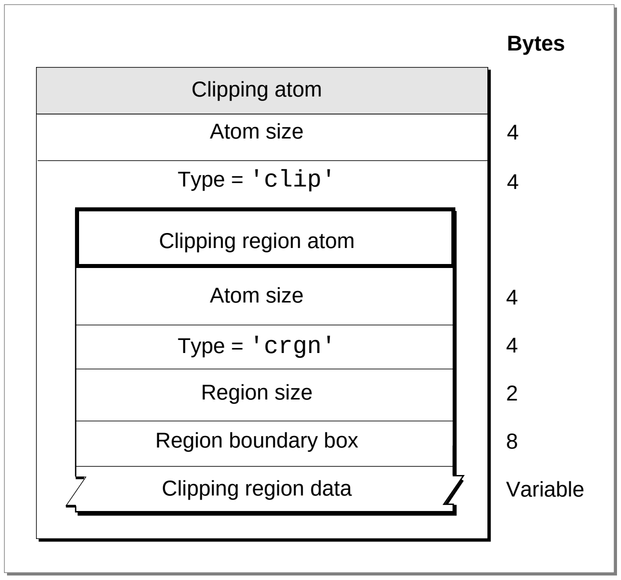
Track width	A 32-bit fixed-point number that specifies the width of this track in pixels.
Track height	A 32-bit fixed-point number that indicates the height of this track in pixels.

Clipping Atoms

Clipping atoms specify the clipping regions for movies and for tracks. The clipping atom has an atom type value of `'clip'`.

Figure 1-11 shows the layout of a clipping atom.

Figure 1-11 The layout of a clipping atom



Clipping atoms contain the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this clipping atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'clip'`.
Clipping region atom	The clipping region atom; these atoms are described in the next section.

Clipping Region Atoms

The clipping region atom contains the data that specifies the clipping region, including its size and bounding box and region. Clipping region atoms have an atom type value of `'crgn'`.

The layout of the clipping region atom is shown in Figure 1-11.

Clipping region atoms contain the following data elements.

Field descriptions

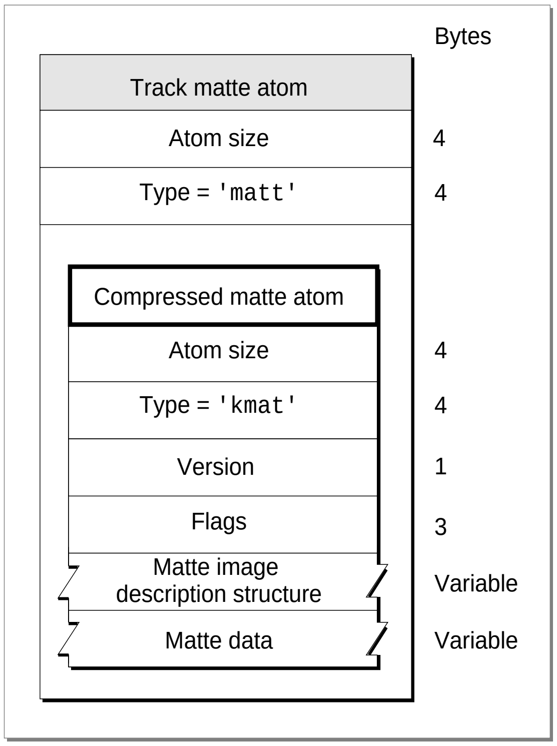
Size	A 32-bit integer that specifies the number of bytes in this clipping region atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'crgn'`.
Region size	
Region boundary box	
Clipping region data	The Region size, Region boundary box, and Clipping region data fields constitute a QuickDraw region. See *Inside Macintosh: Imaging* for details on QuickDraw regions.

Track Matte Atoms

Track matte atoms are used to visually blend the track's image when it is displayed. For a complete description on the use of mattes in QuickTime, see *Inside Macintosh: QuickTime*. Track matte atoms have an atom type value of `'matt'`.

Figure 1-12 shows the layout of a track matte atom.

Figure 1-12 The layout of a track matte atom



Track matte atoms contain the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this track matte atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'matt'`.
Compressed matte atom	The actual matte data in a compressed matte atom. These atoms are described in the next section.

Compressed Matte Atoms

The compressed matte atom specifies the image description structure associated with a particular matte atom. Compressed matte atoms have an atom type value of `'kmat'`.

The layout of the compressed matte atom is shown in Figure 1-12.

Compressed matte atoms contain the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this compressed matte atom.

Type	A 32-bit integer that identifies the atom type; this field must be set to `'kmat'`.
Version	A 1-byte specification of the version of this compressed matte atom.
Flags	Three bytes of space for flags. Set this field to 0.
Matte image description	An image description structure associated with this matte data. The image description contains detailed information that governs how the matte data is used. See "Video Sample Description" on page 59 for more information about image descriptions.
Matte data	The compressed matte data that is of variable length.

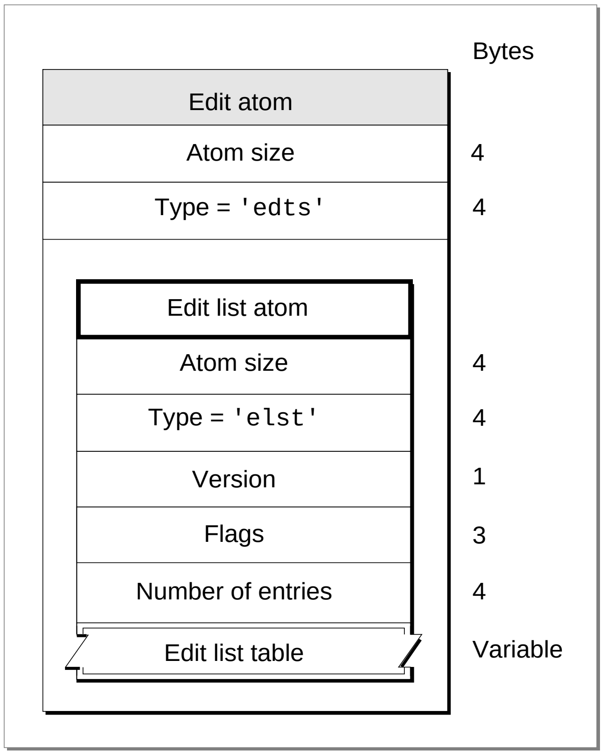
Edit Atoms

You use edit atoms to define the portions of the media that are to be used to build up a track for a movie. The edits themselves are contained in an edit list table, that consists of time offset and duration values for each segment. Edit atoms have an atom type value of `'edts'`.

Figure 1-13 shows the layout of an edit atom.

Note
If the edit atom or the edit list atom is missing, you can assume that the entire media is used by the track. ◆

Figure 1-13 The layout of an edit atom



Edit atoms contain the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this edit atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'edts'`.
Edit list	The edit list atom that contains the edit list information; these atoms are described in the next section.

Edit List Atoms

You use the edit list atom, shown in Figure 1-13, to map from a time in a movie to a time in a media, and ultimately to media data. This information is in the form of an edit list table, shown in Figure 1-14. Edit list atoms have an atom type value of `'elst'`.

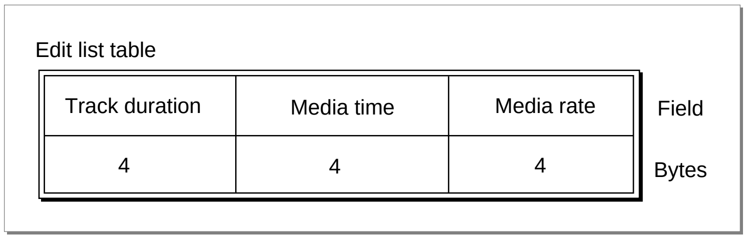
Edit list atoms contain the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this edit list atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'elst'`.

Version	A 1-byte specification of the version of this edit list atom.
Flags	Three bytes of space for flags. Set this field to 0.
Number of entries	A 32-bit integer that specifies the number of entries in the edit list atom that follows.
Edit list table	An array of 32-bit values grouped into entries containing 3 values each.

Figure 1-14 The layout of an edit list table



An edit list table contains the following elements.

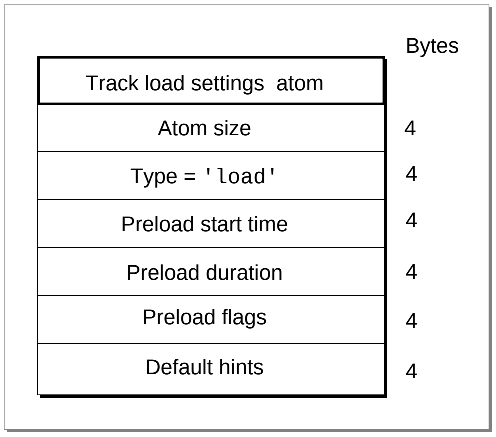
Field descriptions

Track duration	A 32-bit integer that specifies the duration of this edit segment in units of the movie's time scale.
Media time	A 32-bit integer containing the starting time within the media of this edit segment (in media time scale units). If this field is set to –1, it is an empty edit. The last edit in a track should never be an empty edit. Any difference between the movie's duration and the track's duration is expressed as an implicit empty edit.
Media rate	A 32-bit fixed-point number that specifies the relative rate at which to play the media corresponding to this edit segment. This rate value cannot be 0 or negative.

Track Load Settings Atoms

Track load settings atoms contain information that indicates how the track is to be used in its movie. Applications that read QuickTime files can use this information to process the movie data more efficiently. Track load settings atoms are optional and have an atom type value of `'load'`.

Figure 1-15 The layout of a track load settings atom



Track load settings atoms contain the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this track load settings atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'load'`.
Preload start time	A 32-bit integer specifying the starting time, in the movie's time coordinate system, of a segment of the track that is to be preloaded. Used in conjunction with the Preload duration value.
Preload duration	A 32-bit integer specifying the duration, in the movie's time coordinate system, of a segment of the track that is to be preloaded. If the duration is set to –1, it means that the preload segment extends from the preload start time to the end of the track. All media data in the segment of the track defined by the preload start time and preload duration values should be loaded into memory when the movie is to be played.
Preload flags	A 32-bit integer containing flags governing the preload operation. Only two flags are defined and they are mutually exclusive. If Preload flags is set to 1, the track is to be preloaded regardless of whether it is enabled. If Preload flags is set to 2, the track is only preloaded if it is enabled.
Default hints	A 32-bit integer containing playback hints. More than one flag may be enabled. Flags are enabled by setting them to 1. The following flags are defined.

Double buffer	This flag indicates that the track should be played using double-buffered I/O. This flag's value is 0x0020.
High quality	This flag indicates that the track should be displayed at highest possible quality, without regard to real time performance considerations. This flag's value is 0x0100.

Track Reference Atoms

Track reference atoms define relationships between tracks. Track reference atoms allow tracks to specify their relationships to other tracks. For example, if a movie has three video tracks and three sound tracks, track references allow you to identify the related sound and video tracks. Track reference atoms have an atom type value of `'tref'`.

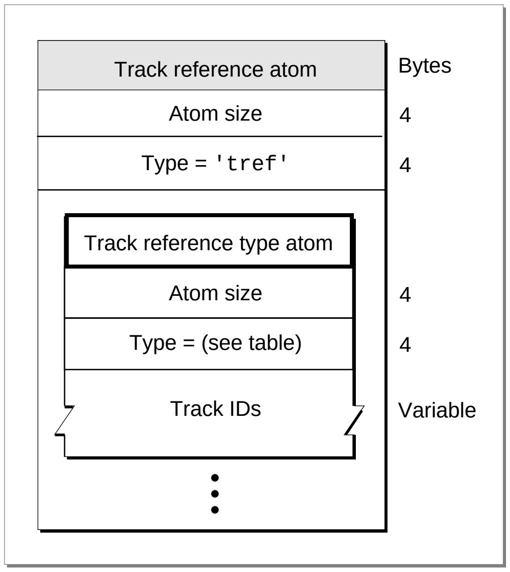
Track references are uni-directional and point from the recipient track to the source track. For example, a video track may reference a time code track to indicate where its time code is stored, but the time code track would not reference the video track. The time code track is the source of time information for the video track.

A single track may reference multiple tracks. For example, a video track could reference a sound track to indicate that the two are synchronized and a time code track to indicate where its time code is stored.

A single track may also be referenced by multiple tracks. For example, both a sound and video track could reference the same time code track if they share the same timing information.

Figure 1-16 shows the layout of a track reference atom.

Figure 1-16 The layout of a track reference atom



A track reference atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this track reference atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'tref'`.

Reference atoms	A list of track reference type atoms containing the track reference information. These atoms are described next.

Each track reference atom defines relationships with tracks of a specific type. The reference type implies a track type. Table 1-2 shows the track reference types and their descriptions.

Table 1-2 Track reference types

Reference type	Description
`'tmcd'`	Time code. Usually references a time code track.
`'chap'`	Chapter or scene list. Usually references a text track.
`'sync'`	Synchronization. Usually between a video and sound track. Indicates that the two tracks are synchronized. The reference can be from either track to the other, or there may be two references.
`'scpt'`	Transcript. Usually references a text track.
`'ssrc'`	Non-primary source. Indicates that the referenced track should send its data to this track, rather than presenting it. The referencing track will use the data to modify how it presents its data. See the next section, "Track Input Map Atoms," for more information.

Each track reference type atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this track reference type atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to one of the values shown in Table 1-2.
Track IDs	A list of track ID values specifying the related tracks. Note that this is one case where track ID values may be set to 0. Unused entries in the atom may have a track ID value of 0. Setting the track ID to 0 may be more convenient that deleting the reference.

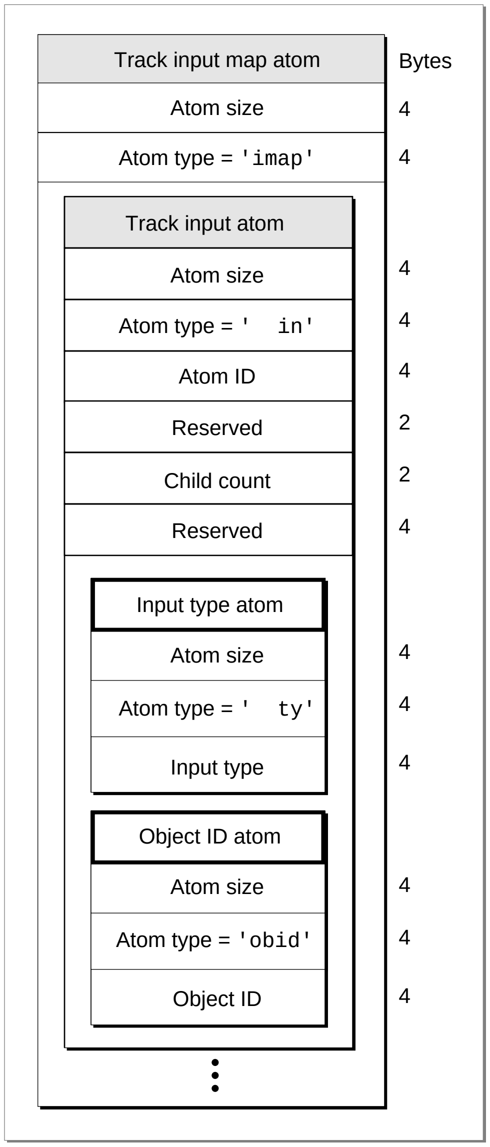
You can determine the number of track references stored in a track reference type atom by subtracting its header size from its overall size and then dividing by the size, in bytes, of a track ID.

Track Input Map Atoms

Track input map atoms define how data being sent to this track from its non-primary sources is to be interpreted. Track references of type `'ssrc'` define a track's secondary data sources. These sources provide additional data that is used when processing the track. Track input map atoms are optional and have an atom type value of `'imap'`.

Figure 1-17 shows the layout of a track input map atom. This atom contains one or more track input atoms. Note that the track input map atom is a QT atom structure.

Figure 1-17 The layout of a track input map atom



Each track input map atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this track input map atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'imap'`.
Track input atoms	A list of track input atoms specifying how to use the input data.

The input map defines all of the track's secondary inputs. Each secondary input is defined using a separate track input atom.

Each track input atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this track input atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'  in'` (note that the two leading bytes must be set to 0x00).
Atom ID	A 32-bit integer relating this track input atom to its secondary input. The value of this field corresponds to the index of the secondary input in the track reference atom. That is, the first secondary input corresponds to the track input atom with an Atom ID value of 1; the second to the track input atom with an Atom ID of 2, and so on.
Reserved	A 16-bit integer that must be set to 0.
Child count	A 16-bit integer specifying the number of child atoms in this atom.
Reserved	A 32-bit integer that must be set to 0.

The track input atom, in turn, may contain two other types of atoms: input type atoms and object ID atoms. The input type atom is required; it specifies how the data is to be interpreted.

The input type atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this input type atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'  ty'` (note that the two leading bytes must be set to 0x00).
Input type	A 32-bit integer that specifies the type of data that is to be received from the secondary data source. Table 1-3 lists valid values for this field.

Table 1-3 Input types

Input identifier	Description
1	A 3x3 transformation matrix to transform the track's location, scaling, and so on.
2	A QuickDraw clipping region to change the track's shape.
3	An 8.8 fixed point value indicating the relative sound volume. This is used for fading the volume.
4	A 16-bit integer indicating the sound balance level. This is used for panning the sound location.
5	A graphics mode record (32-bit integer indicating graphics mode, followed by an RGB color) to modify the track's graphics mode for visual fades.

Table 1-3 Input types (continued)

Input identifier	Description
6	A 3x3 transformation matrix to transform an object within the track's location, scaling, and so on.
7	A graphics mode record (32-bit integer indicating graphics mode, followed by an RGB color) to modify an object within the track's graphics mode for visual fades.
`'vide'`	Compressed image data for an object within the track.

If the input is operating on an object within the track (for example, a sprite within a sprite track), an object ID atom must be included in the track input atom to identify the object.

The object ID atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this object ID atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'obid'`.
Object ID	A 32-bit integer identifying the object.

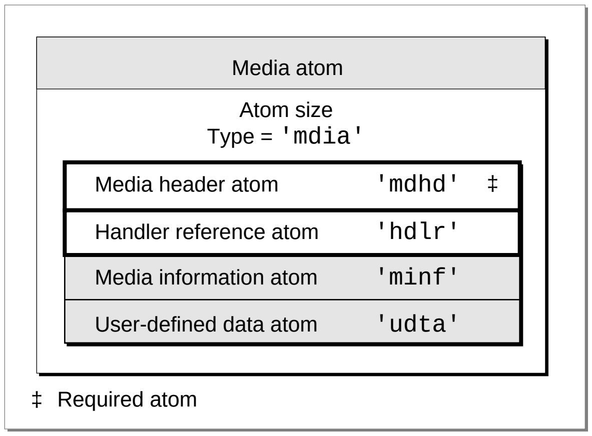
Media Atoms

Media atoms define a track's movie data. The media atom contains information that specifies the media handler component that is to interpret the media data, and it also specifies the data references.

The media atom has an atom type of `'mdia'`. It may contain other atoms, such as a media header (`'mdhd'`), a handler reference (`'hdlr'`), media information (`'minf'`), and user data (`'udta'`). The only required atom in a media atom is the media header atom.

Figure 1-18 shows the layout of a media atom.

Figure 1-18 The layout of a media atom


Media atom
Atom size
Type = 'mdia'
Media header atom 'mdhd' ‡
Handler reference atom 'hdlr'
Media information atom 'minf'
User-defined data atom 'udta'
‡ Required atom

Note
The handler reference atom tells you the kind of media this media atom contains—for example, video or sound. The layout of the media information atom is specific to the media handler that is to interpret the media. "Media Information Atoms," beginning on page 34, discusses how data may be stored in a media, using the video media format defined by Apple as an example. ◆

Media atoms contain the following data elements.

Field descriptions

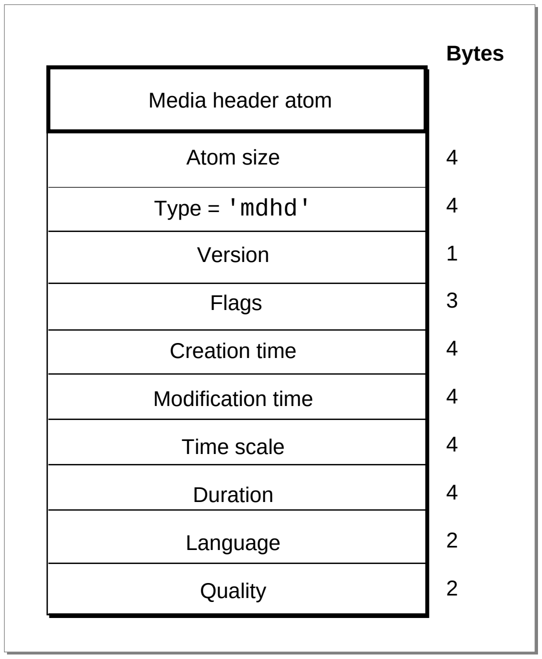
Size	A 32-bit integer that specifies the number of bytes in this media atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'mdia'`.
Media header	The media header atom. This atom contains the standard media information. See the next section for details.
Media handler	The media handler atom. This atom identifies the media handler component that is to be used to interpret the media data. See "Handler Reference Atoms," beginning on page 33, for more information.
Media information	The media information atom. This atom contains media-type specific data for use by the media handler component. See "Media Information Atoms" beginning on page 34 for more information.
User data	The user data atom associated with this media. See "User Data Atoms" on page 57 for more information.

Media Header Atoms

The media header atom specifies the characteristics of a media, including time scale and duration. The media atom has an atom type of `'mdhd'`.

Figure 1-19 shows the layout of a media header atom.

Figure 1-19 The layout of a media header atom



Media header atoms contain the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this media header atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'mdhd'`.
Version	One byte that specifies the version of this movie.
Flags	Three bytes of space for media header flags. Set this field to 0.
Creation time	A 32-bit integer that specifies (in seconds since midnight, January 1, 1904) when the media atom was created.
Modification time	A 32-bit integer that specifies (in seconds since midnight, January 1, 1904) when the media atom was changed.
Time scale	A time value that indicates the time scale for this media—that is, the number of time units that pass per second in its time coordinate system.
Duration	The duration of this media in units of its time scale.
Language	A 16-bit integer that specifies the language code for this media. See "Basic Data Types" on page 75 for valid language codes.

Quality	A 16-bit integer that specifies the media's playback quality—that is, its suitability for playback in a given environment. See *Inside Macintosh: QuickTime* for details on playback quality.

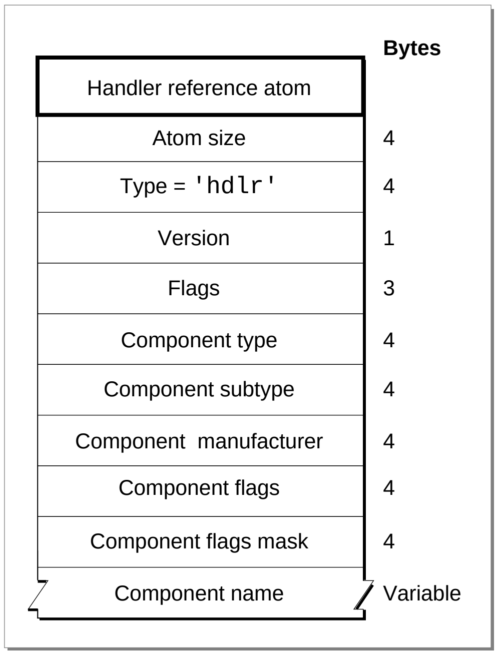
Handler Reference Atoms

The handler reference atom specifies the media handler component that is to be used to interpret the media's data. The handler reference atom has an atom type value of `'hdlr'`.

Historically, the handler reference atom was also used for data references. However, this use may now be ignored.

Figure 1-20 shows the layout of a handler reference atom.

Figure 1-20 The layout of a handler reference atom



Handler reference atoms contain the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this handler reference atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'hdlr'`.
Version	A 1-byte specification of the version of this handler information.
Flags	A 3-byte space for handler information flags. Set this field to 0.

Component type	A four-character code that identifies the type of the handler. Only two values are valid for this field: `'mhlr'` for media handlers and `'dhlr'` for data references.
Component subtype	A four-character code that identifies the type of the media handler or data handler. For media handlers, this field defines the type of data, for example, `'vide'` for video data or `'soun'` for sound data. For data handlers, this field defines the data reference type. For example, a Component subtype value of `'alis'` identifies a file alias.
Component manufacturer	Reserved. Set to 0.
Component flags	Reserved. Set to 0.
Component flags mask	Reserved. Set to 0.
Component name	A Pascal string that specifies the name of the component—that is, the media handler used when this movie was created. This field may contain a zero-length (empty) string.

Media Information Atoms

Media information atoms (defined by the `'minf'` atom type) store handler-specific information for a track's media data. The media handler uses this information to map from media time to media data and to process the media data.

These atoms contain information that is specific to the type of data defined by the media. Further, the format and content of media information atoms are dictated by the media handler that is responsible for interpreting the media data stream. Another media handler would not know how to interpret this information.

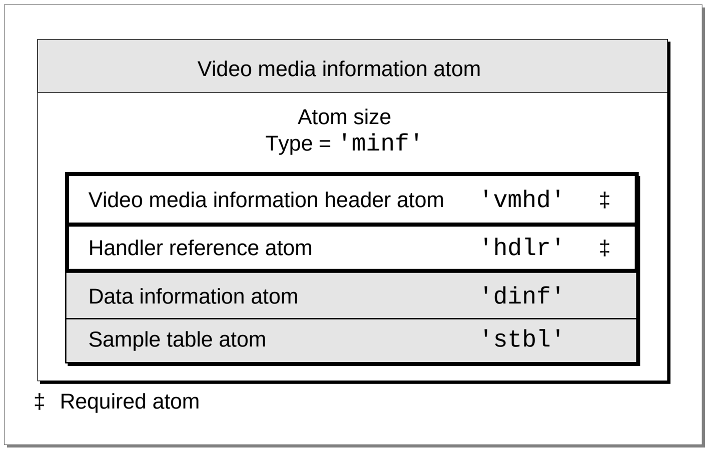
This section describes the atoms that store media information for the video (defined by the `'vmhd'` atom type), sound (defined by the `'smhd'` atom type), and base (defined by the `'gmhd'` atom type) portions of QuickTime movies.

Note
"Using Sample Atoms," beginning on page 56, discusses how the video media handler locates samples in a video media. ◆

Video Media Information Atoms

Video media information atoms are the highest-level atoms in video media. These atoms contain a number of other atoms that define specific characteristics of the video media data. Figure 1-21 shows the layout of a video media information atom.

Figure 1-21 The layout of a media information atom for video


Video media information atom
Atom size
Type = 'minf'
Video media information header atom 'vmhd' ‡
Handler reference atom 'hdlr' ‡
Data information atom 'dinf'
Sample table atom 'stbl'
‡ Required atom

The video media information atom contains the following data elements.

Field descriptions

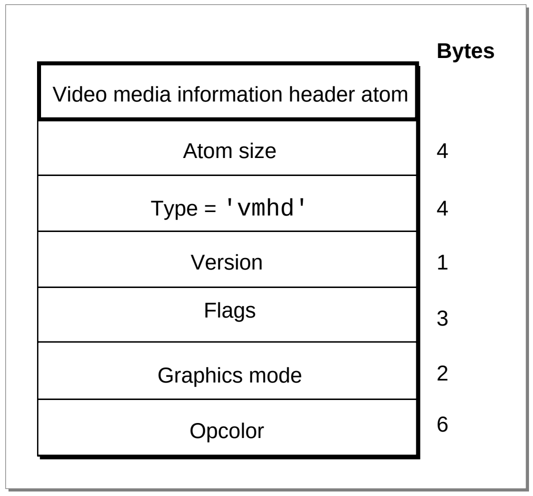
Size	A 32-bit integer that specifies the number of bytes in this video media information atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'minf'`.
Video media information	The video media information header atom (a required atom), which is described in the next section.
Handler reference	The handler reference atom (a required atom), which contains information specifying the data handler component that provides access to the media data. "Handler Reference Atoms" beginning on page 33 discusses handler reference atoms. The handler uses the data information atom to understand the media's data references.
Data information	The data information atom, described in "Data Information Atoms" on page 41.
Sample table	The sample table atom, described in "Sample Table Atoms" on page 45.

Video Media Information Header Atoms

Video media information header atoms define specific color and graphics mode information.

Figure 1-22 shows the structure of a video media information header atom.

Figure 1-22 The layout of a media information header atom for video



The video media information header atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this video media information header atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'vmhd'`.
Version	A 1-byte specification of the version of this video media information header atom.
Flags	A 3-byte space for video media information flags. There is one defined flag.

No lean ahead

This is a compatibility flag that allows QuickTime to distinguish between movies created for QuickTime 1.0 and newer movies. You should always set this flag to 1, unless you are creating a movie intended for playback using version 1.0 of QuickTime. This flag's value is 0x0001.

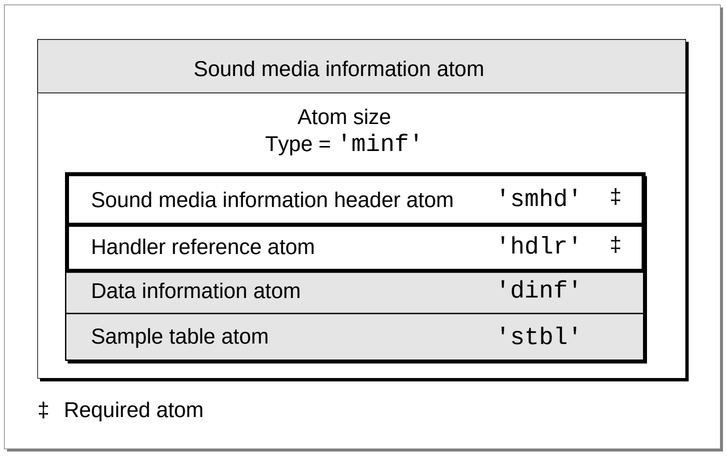
Graphics mode	A 16-bit integer that specifies the transfer mode. The transfer mode specifies which Boolean operation QuickDraw should perform when drawing or transferring an image from one location to another.
Opcolor	Three 16-bit values that specify the red, green, and blue colors for the transfer mode operation indicated in the Graphics mode field.

For comprehensive details on transfer modes and opcolors and their values in QuickDraw, see *Inside Macintosh: Imaging*.

Sound Media Information Atoms

Sound media information atoms are the highest-level atoms in sound media. These atoms contain a number of other atoms that define specific characteristics of the sound media data. Figure 1-23 shows the layout of a sound media information atom.

Figure 1-23 The layout of a media information atom for sound


Sound media information atom
Atom size
Type = 'minf'
Sound media information header atom 'smhd' ‡
Handler reference atom 'hdlr' ‡
Data information atom 'dinf'
Sample table atom 'stbl'
‡ Required atom

The sound media information atom contains the following data elements.

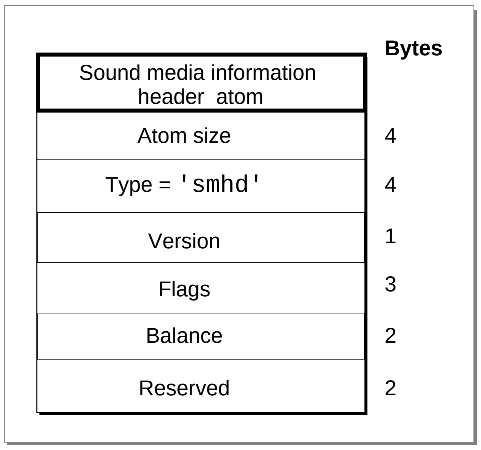
Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this sound media information atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'minf'`.
Sound media information	The sound media information header atom (a required atom), which is described in the next section.
Handler reference	The handler reference atom (a required atom), which contains information specifying the data handler component that provides access to the media data. “Handler Reference Atoms” beginning on page 33 discusses handler reference atoms. The handler uses the data information atom to understand the media’s data references.
Data information	The data information atom, described in “Data Information Atoms” on page 41.
Sample table	The sample table atom, described in “Sample Table Atoms” on page 45.

Sound Media Information Header Atoms

The sound media information header atom (shown in Figure 1-24) stores the sound media’s control information, such as balance.

Figure 1-24 The layout of a sound media information header atom



The sound media information header atom contains the following data elements.

Field descriptions

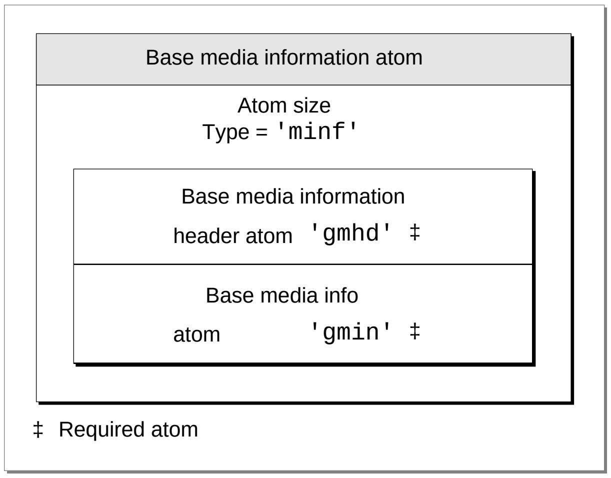
Size	A 32-bit integer that specifies the number of bytes in this sound media information header atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'smhd'`.
Version	A 1-byte specification of the version of this sound media information header atom.
Flags	A 3-byte space for sound media information flags. Set this field to 0.
Balance	A 16-bit integer that specifies the sound balance of this sound media. Sound balance is the setting that controls the mix of sound between the two speakers of a computer. This field is normally set to 0. See *Inside Macintosh: QuickTime* for more information about QuickTime sound balance values.
Reserved	Reserved for use by Apple. Set this field to 0.

Base Media Information Atoms

The base media information atom (shown in Figure 1-25) stores the media information for media types, such as text, MPEG, time code, and music.

Media types that are derived from the base media handler may add other atoms within the base media information atom, as appropriate. At present, the only media type that defines any additional atoms is the time code media. See “Time Code Media Information Atom” beginning on page 66 for more information about time code media.

Figure 1-25 The layout of the base media information atom


Base media information atom
Atom size
Type = 'minf'
Base media information header atom 'gmhd' ‡
Base media info atom 'gmin' ‡
‡ Required atom

The base media information atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this base media information atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'minf'`.
Base media information header	The base media information header atom (a required atom), which is described in the next section.
Base media info	The base media info atom contains control information that describes the media.

Base Media Information Header Atoms

The base media information header atom indicates that this media information atom pertains to a base media.

The base media information header atom contains the following data elements.

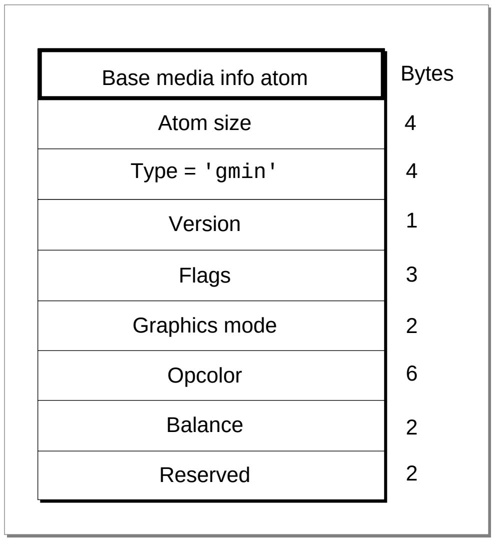
Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this base media information header atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'gmhd'`.

Base Media Info Atoms

The base media info atom, contained in the base media information atom, defines the media's control information, including graphics mode and balance information. Figure 1-26 shows the layout of a base media info atom.

Figure 1-26 The layout of the base media info atom



The base media info atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this base media information header atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'gmin'`.
Version	A 1-byte specification of the version of this base media information header atom.
Flags	A 3-byte space for base media information flags. Set this field to 0.
Graphics mode	A 16-bit integer that specifies the transfer mode. The transfer mode specifies which Boolean operation QuickDraw should perform when drawing or transferring an image from one location to another.
Opcolor	Three 16-bit values that specify the red, green, and blue colors for the transfer mode operation indicated in the Graphics mode field.
Balance	A 16-bit integer that specifies the sound balance of this media. Sound balance is the setting that controls the mix of sound between the two speakers of a computer. This field is normally set to 0. See *Inside Macintosh: QuickTime* for more information about QuickTime sound balance values.
Reserved	Reserved for use by Apple. Set this field to 0.

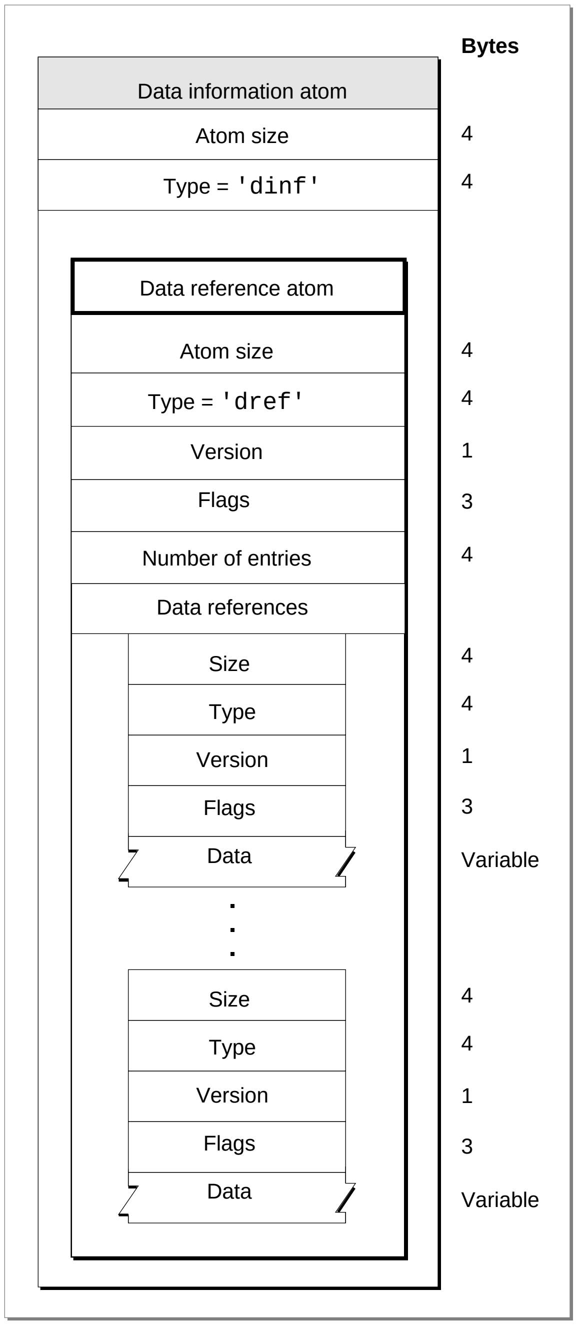
For comprehensive details on transfer modes and opcolors and their values in QuickDraw, see *Inside Macintosh: Imaging*.

Data Information Atoms

The handler reference atom (described in "Handler Reference Atoms," beginning on page 33) contains information specifying the data handler component that provides access to the media data. The data handler component uses the data information atom to interpret the media's data. Data information atoms have an atom type value of `'dinf'`.

Figure 1-27 shows the layout of a data information atom.

Figure 1-27 The layout of a data information atom



The data information atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this data information atom.

Type	A 32-bit integer that identifies the atom type; this field must be set to `'dinf'`.
Data references	A data reference atom, described in the next section, contains the data references.

Data Reference Atoms

Data reference atoms contain tabular data that instructs the data handler component how to access the media's data. Figure 1-27 shows the data reference atom.

The data reference atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this data reference atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'dref'`.
Version	A 1-byte specification of the version of this data reference atom.
Flags	A 3-byte space for data reference flags. Set this field to 0.
Number of entries	A 32-bit integer containing the count of data references that follow.
Data references	An array of data references.

Each data reference is formatted like an atom and contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in these data references.
Type	A 32-bit integer that specifies the type of the data in the data references. Table 1-4 lists valid Type values.
Version	A 1-byte specification of the version of these data references.
Flags	A 3-byte space for data reference flags. There is one defined flag. Self reference This flag indicates that the media's data is in the same file as the movie atom. On the Macintosh, and other file systems with multifork files, set this flag to 1 even if the data resides in a different fork from the movie atom. This flag's value is 0x0001.
Data references	The data reference information.

Table 1-4 shows the currently defined data reference types that may be stored in a movie.

Table 1-4 Data reference types

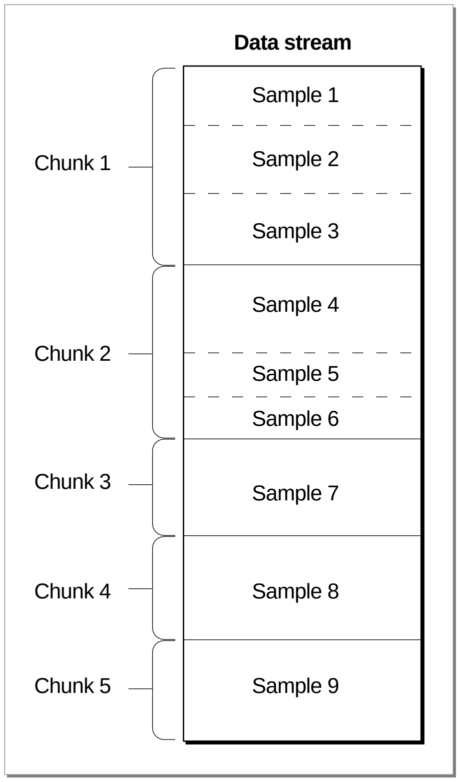
Data reference type	Description
`'alis'`	Data reference is a Macintosh alias. An alias contains information about the file, including its full path name. For more information, see *Inside Macintosh: Files*.
`'rsrc'`	Data reference is a Macintosh alias. Appended to the end of the alias is the resource type (stored as a 32-bit unsigned integer) and ID (stored as a 16-bit signed integer) to use within the specified file.

Sample Atoms

QuickTime stores media data in samples. A sample is a single element in a sequence of time-ordered data. Samples are stored in the media, and they may have varying durations.

Figure 1-28 shows the way that samples are stored in a series of **chunks** in a media.

Figure 1-28 Samples in a media



Chunks are a collection of data samples in a media that allow optimized data access. A chunk may contain one or more samples. Chunks in a media may have different sizes, and the individual samples within a chunk may have different sizes from one another.

One way to describe a sample is to use a sample table atom. The sample table atom acts as a storehouse of information about the samples and contains a number of different types of atoms. The various atoms contain information that allows the media handler to parse the samples in the proper order. This approach enforces an ordering of the samples without requiring that the sample data be stored sequentially with respect to movie time in the actual data stream.

The next section discusses the sample table atom. Subsequent sections discuss each of the atoms that may reside in a sample table atom.

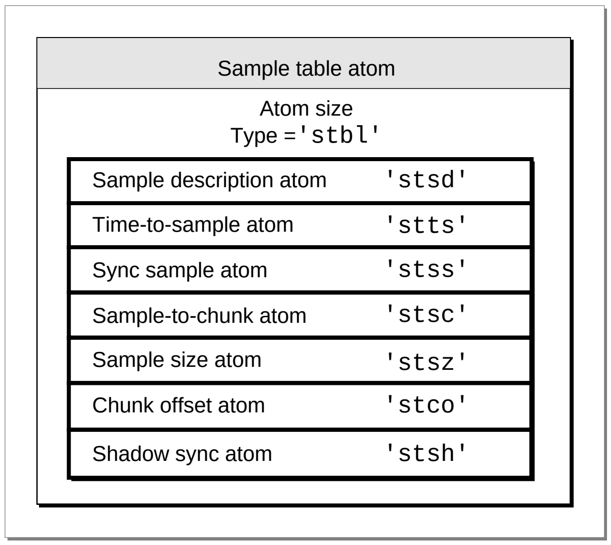
Sample Table Atoms

The sample table atom contains information for converting from media time to sample number to sample location. This atom also indicates how to interpret the sample (for example, whether to decompress the video data and, if so, how). This section describes the format and content of the sample table atom.

The sample table has an atom type of `'stbl'`. It contains the sample description atom, the time-to-sample atom, the sample-to-chunk atom, the sync sample atom, the sample size atom, the chunk offset atom, and the shadow sync atom.

Figure 1-29 shows the layout of a sample table atom.

Figure 1-29 The layout of a sample table atom



The sample table atom contains the following data elements.

Field descriptions

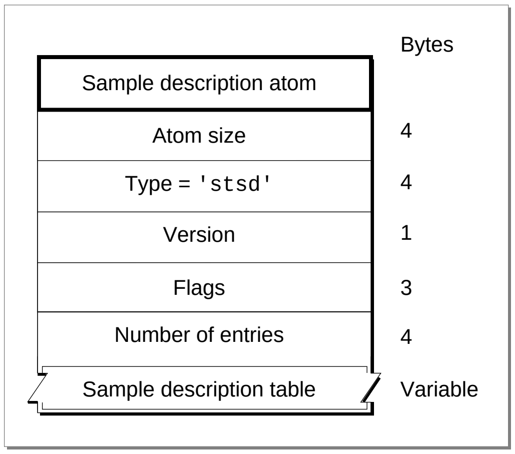
Size	A 32-bit integer that specifies the number of bytes in this sample table atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'stbl'`.
Sample description	The sample description atom, described in the next section.
Time-to-sample	The time-to-sample atom, described in "Time-to-Sample Atoms," beginning on page 48.
Sync sample	The sync sample atom, described in "Sync Sample Atoms," beginning on page 50.
Sample-to-chunk	The sample-to-chunk atom, described in "Sample-to-Chunk Atoms," beginning on page 51.
Sample size	The sample size atom, described in "Sample Size Atoms," beginning on page 53.
Chunk offset	A chunk offset atom, described in "Chunk Offset Atoms," beginning on page 55.
Shadow sync	The shadow sync atom. This atom is obsolete.

Sample Description Atoms

The sample description atom stores information that allows you to decode samples in the media. The data stored in the sample description varies, depending on the media type. For example, in the case of video media, the sample descriptions are image description structures. The sample description information for each media type is explained later in this document, in "Media Data Atom Types" beginning on page 59.

Figure 1-30 shows the layout of a sample description atom.

Figure 1-30 The layout of a sample description atom



The sample description atom has an atom type of `'stsd'`. The sample description atom contains a table of sample descriptions. A media may have one or more sample descriptions, depending upon the number of different encoding schemes used in the media and on the number of files used to store the data. The sample-to-chunk atom identifies the sample description for each sample in the media by specifying the index into this table for the appropriate description (see "Sample-to-Chunk Atoms," beginning on page 51).

The sample description atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this sample description atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'stsd'`.
Version	A 1-byte specification of the version of this sample description atom.
Flags	A 3-byte space for sample description flags. Set this field to 0.
Number of entries	A 32-bit integer containing the number of sample descriptions that follow.

Sample description table	An array of sample descriptions.

While the exact format of the sample description varies by media type, the first four fields of every sample description table are the same.

Field descriptions

Sample description size	A 32-bit integer indicating the number of bytes in the sample description.
Data format	A 32-bit integer indicating the format of the stored data. This depends on the media type, but is usually either the compression format or the media type.
Reserved	Six bytes that must be set to 0.
Data reference index	A 16-bit integer that contains the index of the data reference to use to retrieve data associated with samples that use this sample description. Data references are stored in data reference atoms.

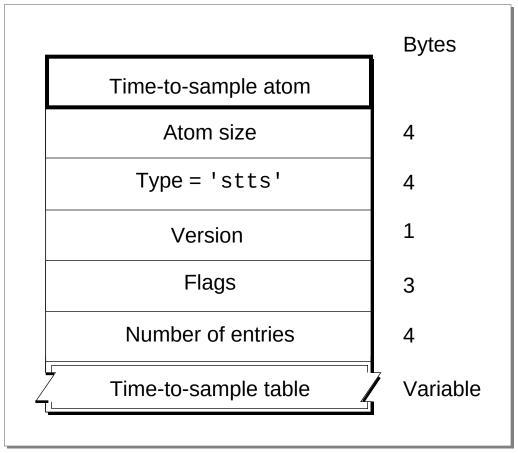
Time-to-Sample Atoms

Time-to-sample atoms store duration information for a media's samples, providing a mapping from a time in the media to the corresponding data sample. The time-to-sample atom has an atom type of `'stts'`.

You can determine the appropriate sample for any time in a media by examining the time-to-sample atom table, which is contained in a time-to-sample atom.

Figure 1-31 shows the layout of a time-to-sample atom.

Figure 1-31 The layout of a time-to-sample atom

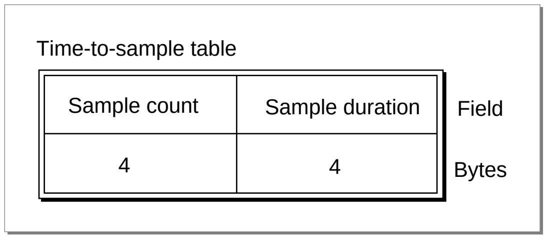


The time-to-sample atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this time-to-sample atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'stts'`.
Version	A 1-byte specification of the version of this time-to-sample atom.
Flags	A 3-byte space for time-to-sample flags. Set this field to 0.
Number of entries	A 32-bit integer containing the number of entries in the time-to-sample table.
Time-to-sample table	A table that defines the duration of each sample in the media. Each table entry contains a count field and a duration field. The structure of a time-to-sample table is shown in Figure 1-32.

Figure 1-32 The layout of a time-to-sample table



You define a time-to-sample table by specifying these entries:

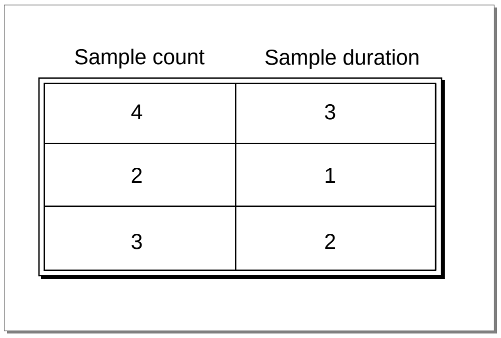
Field descriptions

Sample count	A 32-bit integer that specifies the number of consecutive samples that have the same duration.
Sample duration	A 32-bit integer that specifies the duration of each sample.

Entries in the table describe samples according to their order in the media and their duration. If consecutive samples have the same duration, a single table entry may be used to define more than one sample. In these cases, the count field indicates the number of consecutive samples that have the same duration. For example, if a video media has a constant frame rate, this table would have one entry and the count would be equal to the number of samples.

Figure 1-33 presents an example of a time-to-sample table that is based on the chunked media data shown in Figure 1-28 on page 45. That data stream contains a total of nine samples that correspond in count and duration to the entries in the table shown here. Even though samples 4, 5, and 6 are in the same chunk, sample 4 has a duration of 3, and samples 5 and 6 have a duration of 2.

Figure 1-33 An example of a time-to-sample table



Sample count	Sample duration
4	3
2	1
3	2

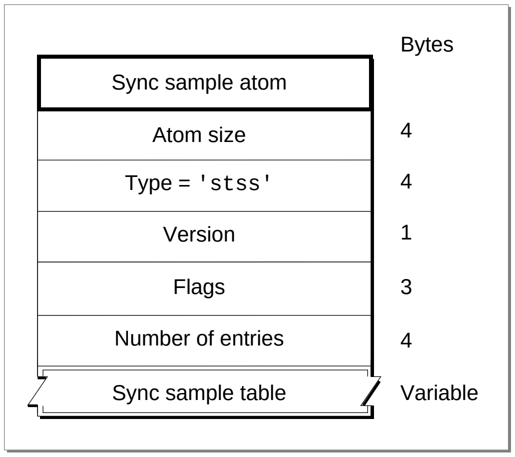
Sync Sample Atoms

The sync sample atom identifies the **key frames** in the media. In a media that contains compressed data, key frames define starting points for portions of a temporally compressed sequence. The key frame is self-contained—that is, it is independent of preceding frames. Subsequent frames may depend on the key frame.

Sync sample atoms have an atom type of `'stss'`. The sync sample atom contains a table of sample numbers. Each entry in the table identifies a sample that is a key frame for the media. Figure 1-34 shows the layout of a sync sample atom.

If no sync sample atom exists, then all the samples are key frames.

Figure 1-34 The layout of a sync sample atom

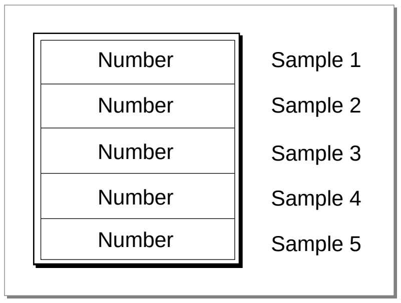


The sync sample atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this sync sample atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'stss'`.
Version	A 1-byte specification of the version of this sync sample atom.
Flags	A 3-byte space for sync sample flags. Set this field to 0.
Number of entries	A 32-bit integer containing the number of entries in the sync sample table.
Sync sample table	A table of sample numbers; each sample number corresponds to a key frame. Figure 1-35 shows the layout of a sync sample table.

Figure 1-35 The layout of a sync sample table

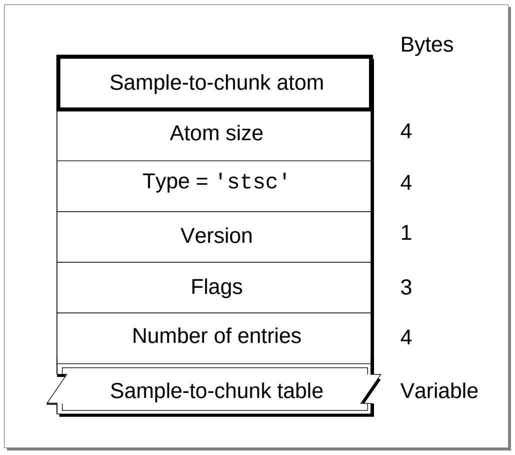


Sample-to-Chunk Atoms

As samples are added to a media, they are collected into chunks that allow optimized data access. A chunk may contain one or more samples. Chunks in a media may have different sizes, and the samples within a chunk may have different sizes. The sample-to-chunk atom stores chunk information for the samples in a media.

Sample-to-chunk atoms have an atom type of `'stsc'`. The sample-to-chunk atom contains a table that maps samples to chunks in the media data stream. Figure 1-36 shows the layout of a sample-to-chunk atom. By examining the sample-to-chunk atom, you can determine the chunk that contains a specific sample.

Figure 1-36 The layout of a sample-to-chunk atom

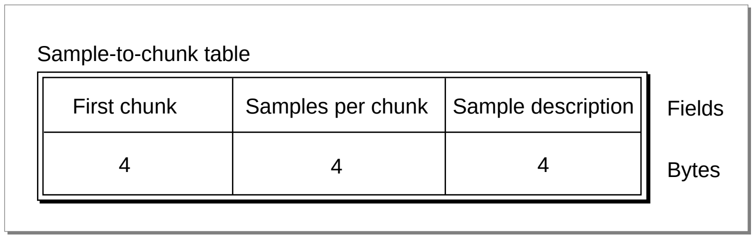


The sample-to-chunk atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this sample-to-chunk atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'stsc'`.
Version	A 1-byte specification of the version of this sample-to-chunk atom.
Flags	A 3-byte space for sample-to-chunk flags. Set this field to 0.
Number of entries	A 32-bit integer containing the number of entries in the sample-to-chunk table.
Sample-to-chunk table	A table that maps samples to chunks. Figure 1-37 shows the structure of a sample-to-chunk table. Each sample-to-chunk atom contains such a table, which identifies the chunk for each sample in a media. Each entry in the table contains a first chunk field, a samples per chunk field, and a sample description ID field. From this information, you can ascertain where samples reside in the media data.

Figure 1-37 The layout of a sample-to-chunk table



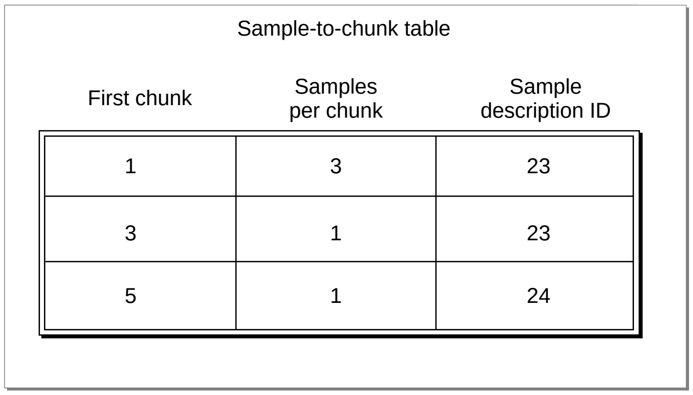
You define a sample-to-chunk table by specifying the following data elements.

Field descriptions

First chunk	The first chunk number using this table entry.
Samples per chunk	The number of samples in each chunk.
Sample description ID	The identification number associated with the sample description for the sample. For details on sample description atoms, see "Sample Description Atoms," beginning on page 47.

Figure 1-38 shows an example of a sample-to-chunk table that is based on the data stream shown in Figure 1-28 on page 45.

Figure 1-38 An example of a sample-to-chunk table



Sample-to-chunk table

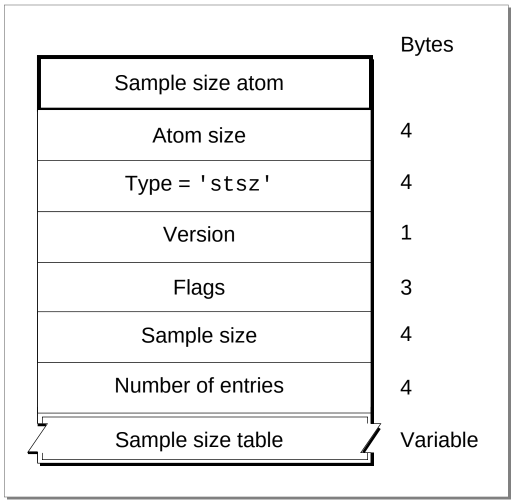
First chunk	Samples per chunk	Sample description ID
1	3	23
3	1	23
5	1	24

Each table entry corresponds to a set of consecutive chunks, each of which contains the same number of samples. Furthermore, each of the samples in these chunks must use the same sample description. Whenever the number of samples per chunk or the sample description changes, you must create a new table entry. If all the chunks have the same number of samples per chunk and use the same sample description, this table has one entry.

Sample Size Atoms

You use sample size atoms to specify the size of each sample in the media. Sample size atoms have an atom type of `'stsz'`. Figure 1-39 shows the layout of a sample size atom.

Figure 1-39 The layout of a sample size atom



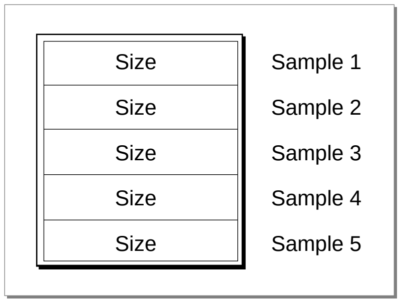
The sample size atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this sample size atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'stsz'`.
Version	A 1-byte specification of the version of this sample size atom.
Flags	A 3-byte space for sample size flags. Set this field to 0.
Sample size	A 32-bit integer specifying the sample size. If all the samples are the same size, this field contains that size value. If this field is set to 0, then the samples have different sizes, and those sizes are stored in the sample size table.
Number of entries	A 32-bit integer containing the number of entries in the sample size table.
Sample size table	A table containing the sample size information. The sample size table contains an entry for every sample in the media's data stream. Each table entry contains a size field. The size field contains the size, in bytes, of the sample in question. The table is indexed by sample number—the first entry corresponds to the first sample, the second entry is for the second sample, and so on.

Figure 1-40 shows a sample size table.

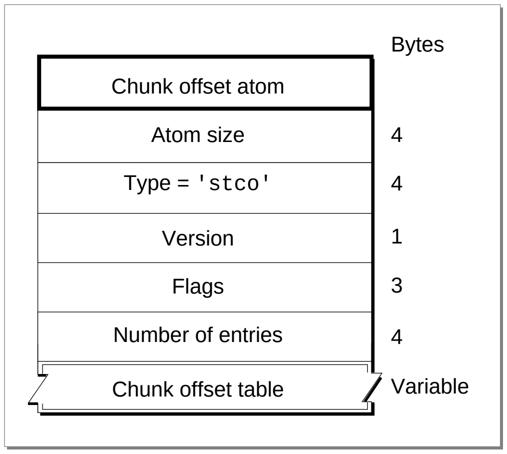
Figure 1-40 An example of a sample size table



Chunk Offset Atoms

Chunk offset atoms identify the location of each chunk of data in the media's data stream. Chunk offset atoms have an atom type of `'stco'`. The chunk offset atom (shown in Figure 1-41) contains a table of offset information.

Figure 1-41 The layout of a chunk offset atom



The chunk offset atom contains the following data elements.

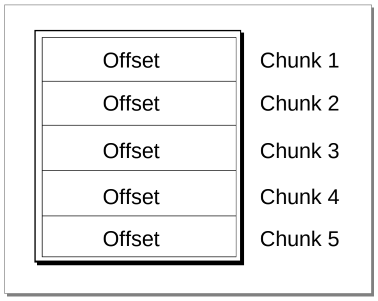
Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this chunk offset atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'stco'`.
Version	A 1-byte specification of the version of this chunk offset atom.
Flags	A 3-byte space for chunk offset flags. Set this field to 0.
Number of entries	A 32-bit integer containing the number of entries in the chunk offset table.

Chunk offset table	A chunk offset table consisting of an array of offset values. There is one table entry for each chunk in the media. The offset contains the byte offset from the beginning of the data stream to the chunk. The table is indexed by chunk number—the first table entry corresponds to the first chunk, the second table entry is for the second chunk, and so on.

Figure 1-42 shows an example of a chunk offset table

Figure 1-42 An example of a chunk offset table



Using Sample Atoms

This section presents examples using the atoms just described. These examples are intended to help you understand the relationships between these atoms. The first section, "Finding a Sample," describes the steps that the video media handler uses to find the sample that contains the media data for a particular time in a media. The second section, "Finding a Key Frame," describes the steps that the video media handler uses to find an appropriate key frame for a specific time in a movie.

Finding a Sample

When QuickTime displays a movie or track, it tells the appropriate media handler to access the media data for a particular time. The media handler must correctly interpret the data stream to retrieve the requested data. In the case of video media, the media handler traverses several atoms to find the location and size of a sample for a given media time. The media handler does the following:

1. Determines the time in the media time coordinate system.
2. Examines the time-to-sample atom to determine the sample number that contains the data for the specified time.
3. Scans the sample-to-chunk atom to discover which chunk contains the sample in question.
4. Extracts the offset to the chunk from the chunk offset atom.
5. Finds the offset within the chunk and the sample's size by using the sample size atom.

Finding a Key Frame

Finding a key frame for a specified time in a movie is slightly more complicated than finding a sample for a specified time. The media handler must use the sync sample atom and the time-to-sample atom together in order to find a key frame. The media handler does the following:

1. Examines the time-to-sample atom to determine the sample number that contains the data for the specified time.
2. Scans the sync sample atom to find the key frame that precedes the sample number chosen in step 1.
3. Scans the sample-to-chunk atom to discover which chunk contains the key frame.
4. Extracts the offset to the chunk from the chunk offset atom.
5. Finds the offset within the chunk and the sample's size by using the sample size atom.

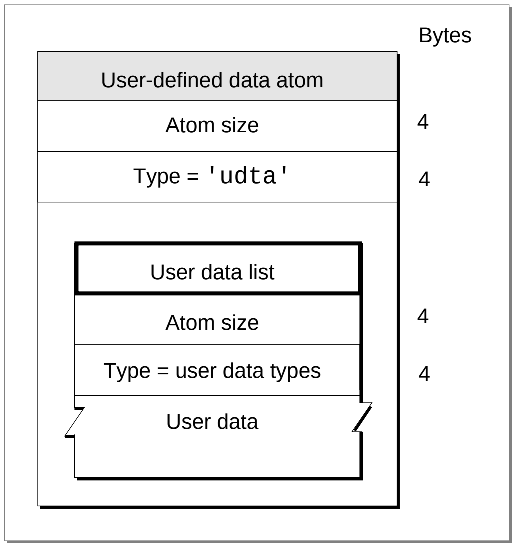
User Data Atoms

User data atoms allow you to define and store arbitrary data associated with a QuickTime object, such as a movie, track, or media. The user data atom has an atom type of `'udta'`.

Inside the user data atom is a list of atoms describing each piece of user data. User data provides a simple way to extend what is stored in a QuickTime movie. For example, you may use user data atoms to store a movie's window position, playback characteristics, or creation information.

Figure 1-43 shows the layout of a user data atom.

Figure 1-43 The layout of a user data atom



The user data atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this user data atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'udta'`.
User data list	A user data list that is itself formatted like a series of atoms. Each data element in the private data portion of the user-defined data atom contains size and type information along with the data. Furthermore, for historical reasons the list of atoms is optionally terminated by a 32-bit integer set to 0. If you are writing a program to read user data atoms, you should allow for the terminating 0. However, if you are writing a program to create user data atoms, you can safely leave out the trailing 0. Table 1-5 lists the currently defined list entry types.

Table 1-5 User data list entry types

List entry type	Description
`'©cpy'`	Copyright statement.
`'©day'`	Date the movie content was created.
`'©dir'`	Name of movie’s director.
`'©ed1'` to `'©ed9'`	Edit dates and descriptions.
`'©fmt'`	Indication of movie format (computer-generated, digitized, and so on).
`'©inf'`	Information about the movie.
`'©prd'`	Name of movie’s producer.
`'©prf'`	Names of performers.
`'©req'`	Special hardware and software requirements.
`'©src'`	Credits for those who provided movie source content.
`'©wrt'`	Name of movie’s writer
`'WLOC'`	Default window location for movie. Two 16 bit values, {x,y}.
`'name'`	Name of object.
`'LOOP'`	Long integer indicating looping style. 0 for none, 1 for looping, 2 for palindrome looping.
`'SelO'`	Play selection only. Byte indicating that only the selected area of the movie be played.
`'AllF'`	Play all frames. Byte indicating that all frames of video should be played, regardless of timing.

All user data list entries whose type begins with the '©' character (ASCII 169), are defined to be international text. These list entries must contain a list of text strings with associated language codes. By storing multiple versions of the same text, a single user data text item can contain translations for different languages.

The list of text strings uses a small integer atom format, which is identical to the QuickTime atom format, except that it uses 16-bit values for size and type instead of 32-bit values. The first value is the size of the string, including the size and type, and the second value is the language code for the string.

Media Data Atom Types

QuickTime uses atoms of different types to store different types of media data—video media for video data, sound media for audio data, and so on. The following sections discuss each of these different media data atom types.

Video Media

Video media is used to store compressed and uncompressed image data in QuickTime movies. It has a media type of `'vide'`.

Video Sample Description

The video sample description contains information that defines how to interpret video media data. This sample description is based on the standard sample description header, as described in "Sample Table Atoms" beginning on page 45.

The Data format field indicates the type of compression that was used to compress the image data. Table 1-6 shows some of the formats currently supported.

Table 1-6 Image compression formats

Compression type	Description
`'cvid'`	Cinepak
`'jpeg'`	JPEG
`'raw '`	Uncompressed RGB
`'YUV2'`	Uncompressed YUV422
`'smc '`	Graphics
`'rle '`	Animation
`'rpza'`	Apple Video
`'kpcd'`	Kodak Photo CD
`'qdgx'`	QuickDraw GX

Table 1-6 Image compression formats (continued)

Compression type	Description
`'mpeg'`	MPEG still image
`'mjpa'`	Motion-JPEG (Format A)
`'mjpb'`	Motion-JPEG (Format B)

The video media handler also adds some of its own fields to the sample description. For more information about each of these fields, see the discussion of image descriptions in *Inside Macintosh: QuickTime*.

Field descriptions

Version	A 16-bit integer indicating the version number of the compressed data. Usually this is set to 0, unless a compressor has changed its data format.
Revision level	A 16-bit integer that must be set to 0.
Vendor	A 32-bit integer that specifies the developer of the compressor that generated the compressed data. Often this field contains `'appl'` to indicate Apple Computer, Inc.
Temporal quality	A 32-bit integer containing a value from 0 to 1023 indicating the degree of temporal compression.
Spatial quality	A 32-bit integer containing a value from 0 to 1023 indicating the degree of spatial compression.
Width	A 16-bit integer that specifies the width of the source image in pixels.
Height	A 16-bit integer that specifies the height of the source image in pixels.
Horizontal resolution	A 32-bit fixed point number containing the horizontal resolution of the image in pixels per inch.
Vertical resolution	A 32-bit fixed point number containing the vertical resolution of the image in pixels per inch.
Data size	A 32-bit integer that must be set to 0.
Frame count	A 16-bit integer that indicates how many frames of compressed data are stored in each sample. Usually set to 1.
Compressor name	A 32-byte Pascal string containing the name of compressor that created the image, such as `"jpeg"`.
Depth	A 16-bit integer that indicates the pixel depth of the compressed image. Values of 1, 2, 4, 8,16, 24, and 32 indicate the depth of color images. The value of 32 should be used only if the image contains an alpha channel. Values of 34, 36, and 40 indicate 2-, 4-, and 8-bit grayscale, respectively, for grayscale images.
Color table ID	A 16-bit integer that identifies which color table to use. If this field is set to –1, the default color table should be used for the specified depth. For all depths below 16 bits per pixel, this indicates a

standard Macintosh color table for the specified depth. Depths of 16, 24, and 32 have no color table.

If the color table ID is set to 0, a color table is contained within the sample description itself. The color table immediately follows the Color table ID field in the sample description. See “Color Table Atoms” on page 14 for a complete description of a color table.

Video sample descriptions can be extended by appending other atoms. These atoms are placed after the color table, if one is present. These extensions to the sample description may contain display hints for the decompressor or may simply carry additional information associated with the images. Table 1-7 lists the currently defined extensions to video sample descriptions.

Table 1-7 Video sample description extensions

Extension type	Description
`'gama'`	A 32-bit fixed point number indicating the gamma level at which the image was captured. The decompressor can use this value to gamma-correct at display time.
`'fiel'`	A 2-byte value indicating the number of video fields in the data stream, and how those fields are to be used. The first byte specifies the field count, and may be set to 1 or 2. The second byte defines the field dominance, as follows: 0–Field dominance unknown 1–Top field is first, temporally 2–Bottom field is first, temporally This information is used by applications that may be modifying decompressed image data, or by decompressor components to determine field display order.
`'mjqt'`	The default quantization table for a Motion JPEG data stream.
`'mjht'`	The default Huffman table for a Motion JPEG data stream.

Video Sample Data

The format of the data stored in video samples is completely dependent on the type of the compressed data stored in the video sample description. The following sections discuss each of the video encoding schemes supported by QuickTime.

Uncompressed RGB

Uncompressed RGB data is stored in a variety of different formats. The format used depends on the Depth field of the video sample description. For all depths, the image data is padded on each scan line to ensure that each scan line begins on an even byte boundary.

- For depths of 1, 2, 4, and 8, the values stored are indexes into the color table specified in the Color table id field.
- For a depth of 16, the pixels are stored as 5-5-5 RGB values with the high bit of each 16-bit integer set to 0.
- For a depth of 24, the pixels are stored packed together in RGB order.
- For a depth of 32, the pixels are stored with an 8-bit alpha channel, followed by 8-bit RGB components.

Uncompressed YUV

Uncompressed YUV data is stored as YUV422 data. Two pixels are combined into a single 32-bit integer stored in YUYV order.

JPEG

QuickTime stores JPEG images according to the rules described in the ISO JPEG specification, document number DIS 10918-1.

Motion JPEG

Motion JPEG (M-JPEG) is a variant of the ISO JPEG specification for use with digital video streams. Instead of compressing an entire image into a single bitstream, M-JPEG compresses each video field separately, returning the resulting JPEG bitstreams consecutively in a single frame.

There are two flavors of M-JPEG currently in use. These two formats differ based on their use of markers. **M-JPEG format A** supports markers; **M-JPEG format B** does not. The following paragraphs describe how QuickTime stores M-JPEG sample data.

Each field of M-JPEG Format A fully complies with the ISO JPEG specification, and therefore supports application markers. QuickTime uses the app 1 marker to store control information, as follows (all of the fields are 32-bit integers):

Field descriptions

Reserved	Contents unpredictable; should be set to 0.
Tag	Identifies the data type; this field must be set to `'mjpg'`.
Field size	Contains the actual size of the image data for this field, in bytes.
Padded field size	Contains the size of the image data, including pad bytes. Some video hardware may append pad bytes to the image data; this field, along with the Field size field, allows you to compute how many pad bytes were added.
Offset to next field	Specifies the offset, in bytes, from the start of the field data to the start of the next field in the bitstream. This field should be set to 0 in the second field's marker data.
Quantization table offset	Specifies the offset, in bytes, from the start of the field data to the quantization table marker. If this field is set to 0, check the image description for a default quantization table (see "Video Sample Description" beginning on page 59 for details).

Huffman table offset	Specifies the offset, in bytes, from the start of the field data to the Huffman table marker. If this field is set to 0, check the image description for a default Huffman table (see "Video Sample Description" beginning on page 59 for details).
Start of image offset	Specifies the offset from the start of the field data to the start of image marker. This field should never be set to 0.

M-JPEG Format B does not support markers. In place of the marker, therefore, QuickTime inserts a header at the beginning of the bitstream. Again, all of the fields are 32-bit integers.

Field descriptions

Reserved	Contents unpredictable; should be set to 0.
Tag	Identifies the data type; this field must be set to `'mjpg'`.
Field size	Contains the actual size of the image data for this field, in bytes.
Padded field size	Contains the size of the image data, including pad bytes. Some video hardware may append pad bytes to the image data; this field, along with the Field size field, allows you to compute how many pad bytes were added.
Offset to next field	Specifies the offset, in bytes, from the start of the field data to the start of the next field in the bitstream. This field should be set to 0 in the second field's header data.
Quantization table offset	Specifies the offset, in bytes, from the start of the field data to the quantization table. If this field is set to 0, check the image description for a default quantization table (see "Video Sample Description" beginning on page 59 for details).
Huffman table offset	Specifies the offset, in bytes, from the start of the field data to the Huffman table. If this field is set to 0, check the image description for a default Huffman table (see "Video Sample Description" beginning on page 59 for details).
Start of image offset	Specifies the offset from the start of the field data to the field's image data. This field should never be set to 0.
Reserved	Must be set to 0.
Reserved	Must be set to 0.

The M-JPEG Format B header must be a multiple of 16 in size. When you add pad bytes to the header, set them to 0.

Note

Because this format does not support markers, there is no need to stuff the bitstream with null bytes (0x00) after data bytes that are set to 0xFF. ◆

MPEG Still Images

QuickTime stores MPEG still images according to the Blue Book specification for Enhanced CDs.

Sound Media

Sound media is used to store compressed and uncompressed audio data in QuickTime movies. It has a media type of `'soun'`.

Sound Sample Description

The sound sample description contains information that defines how to interpret sound media data. This sample description is based on the standard sample description header, as described in "Sample Table Atoms" beginning on page 45.

The data format field contains the format of the audio data. Table 1-8 shows some of the current formats.

Table 1-8 Sound data format types

Data format	Description
`'raw '`	Samples are stored uncompressed in offset-binary format (values range from 0 to 255; 128 is silence).
`'twos'`	Samples are stored uncompressed, in twos-complement format (sample values range from –128 to 127 for 8-bit audio, and -32768 to 32767 for 1-bit audio; 0 is always silence).
`'MAC3'`	Samples have been compressed using MACE 3:1.
`'MAC6'`	Samples have been compressed using MACE 6:1.
`'ima4'`	Samples have been compressed using IMA 4:1.
`'µlaw'`	Samples have been compressed using µLaw 2:1.

The sound media handler also adds some of its own fields to the sample description. For more information about each of these fields, see *Inside Macintosh: QuickTime*.

Field descriptions

Version	A 16-bit integer that must be set to 0.
Revision level	A 16-bit integer that must be set to 0.
Vendor	A 32-bit integer that must be set to 0.
Number of channels	A 16-bit integer that indicates the number of sound channels used by the sound sample. Set this field to 1 for monaural sounds; set it to 2 for stereo sounds.
Sample size	A 16-bit integer that specifies the number of bits in each uncompressed sound sample. Set this field to 8 for 8-bit sound and

	to 16 for 16-bit sound. Sound stored in `'twos'` format may contain 32-bit samples.
Compression ID	A 16-bit integer that must be set to 0.
Packet size	A 16-bit integer that must be set to 0.
Sample rate	A 32-bit unsigned fixed-point number that indicates the rate at which the sound samples were obtained. This number should match the media's time scale.

Sound Sample Data

The format of data stored in sound samples is completely dependent on the type of the compressed data stored in the sound sample description. The following sections discuss each of the formats supported by QuickTime.

Uncompressed 8-bit Sound

Eight-bit audio may be stored in either twos-complement or offset-binary encodings. In either case, if the data is in stereo, the left and right channels are interleaved.

Uncompressed 16-bit Sound

Sixteen-bit audio may be stored in either twos-complement or offset-binary encodings. However, twos-complement encoding is preferred. In either case, if the data is in stereo, the left and right channels are interleaved.

MACE 3:1 and 6:1

Not currently documented by Apple Computer. These are 8-bit formats.

IMA 4:1

The IMA encoding scheme is based on a standard developed by the International Multimedia Association for pulse code modulation (PCM) audio compression. QuickTime uses a slight variation of the format to allow for random access. IMA is a 16-bit audio format.

µLaw 2:1

The µLaw encoding scheme is used on North American and Japanese phone systems, and is coming into use for voice data interchange, and in PBXs, voice-mail systems, and Internet Talk Radio (via MIME). In µLaw encoding, 14 bits of linear sample data are reduced to 8 bits of logarithmic data.

Time Code Media

Time code media is used to store time code data in QuickTime movies. It has a media type of `'tmcd'`.

Time Code Sample Description

The time code sample description contains information that defines how to interpret time code media data. This sample description is based on the standard sample description header, as described in “Sample Table Atoms” beginning on page 45.

The Data format field in the sample description is always set to `'tmcd'`.

The time code media handler also adds some of its own fields to the sample description.

Field descriptions

Reserved	A 32-bit integer that is reserved for future use. Set this field to 0.
Flags	A 32-bit integer containing flags that identify some time code characteristics. The following flags are defined.

Drop frame	Indicates whether the time code is drop frame. Set it to 1 if the time code is drop frame. This flag’s value is 0x0001.
24 hour max	Indicates whether the time code wraps after 24 hours. Set it to 1 if the time code wraps. This flag’s value is 0x0002.
Negative times OK	Indicates whether negative time values are allowed. Set it to 1 if the time code supports negative values. This flag’s value is 0x0004.
Counter	Indicates whether the time value corresponds to a tape counter value. Set it to 1 if the time code values are tape counter values. This flag’s value is 0x0008.

Time scale	A 32-bit integer that specifies the time scale for interpreting the Frame duration field.
Frame duration	A 32-bit integer that indicates how long each frame lasts in real time.
Number of frames	An 8-bit integer that contains the number of frames per second for the time code format. If the time is a counter, this is the number of frames for each counter tick.
Reserved	A 24-bit quantity that must be set to 0.
Source reference	A user data atom containing information about the source video tape. The only currently used user data list entry is the `'name'` field. This entry contains an international text item specifying the name of the source tape.

Time Code Media Information Atom

The time code media also requires a media information atom. This atom contains information governing how the time code text is displayed. This media information atom is stored in a base media information atom (see “Base Media Information Atoms” beginning on page 38 for more information). The type of the time code media information atom is `'tcmi'`.

The time code media information atom contains the following data elements.

Field descriptions

Size	A 32-bit integer that specifies the number of bytes in this time code media information atom.
Type	A 32-bit integer that identifies the atom type; this field must be set to `'tcmi'`.
Version	A 1-byte specification of the version of this time code media information atom.
Flags	A 3-byte space for time code media information flags. Set this field to 0.
Text font	A 16-bit integer that indicates the font to use. Set this field to 0 to use the system font. If the Font name field contains a valid name, ignore this field.
Text face	A 16-bit integer that indicates the font's style. Set this field to 0 for normal text. You can enable other style options by using one or more of the following bit masks:

0x0001	Bold
0x0002	Italic
0x0004	Underline
0x0008	Outline
0x0010	Shadow
0x0020	Condense
0x0040	Extend

Text size	A 16-bit integer that specifies the point size of the time code text.
Text color	A 48-bit RGB color value for the time code text.
Background color	A 48-bit RGB background color for the time code text.
Font name	A Pascal string specifying the name of the time code text's font.

Time Code Sample Data

There are two different sample data formats used by time code media.

If the Counter flag is set to 1 in the time code sample description, the sample data is a counter value. Each sample contains a 32-bit integer counter value.

If the Counter flag is set to 0 in the time code sample description, the sample data format is a time code record, as follows.

Field descriptions

Hours	An 8-bit unsigned integer that indicates the starting number of hours.
Negative	A 1-bit value indicating the time's sign. If the bit is set to 1, the time code record value is negative.
Minutes	A 7-bit integer that contains the starting number of minutes.
Seconds	An 8-bit unsigned integer indicating the starting number of seconds.

Frames	An 8-bit unsigned integer that specifies the starting number of frames. This field's value cannot exceed the value of the Number of frames field in the time code sample description.

Text Media

Text media is used to store text data in QuickTime movies. It has a media type of `'text'`.

Text Sample Description

The time code sample description contains information that defines how to interpret time code media data. This sample description is based on the standard sample description header, as described in "Sample Table Atoms" beginning on page 45.

The Data format field in the sample description is always set to `'text'`.

The text media handler also adds some of its own fields to the sample description.

Field descriptions

Display flags — A 32-bit integer containing flags that describe how the text should be drawn. The following flags are defined.

Don't auto scale	Controls text scaling. If this flag is set to 1, the text media handler reflows the text instead of scaling when the track is scaled. This flag's value is 0x0002.
Use movie background color	Controls background color. If this flag is set to 1, the text media handler ignores the Background color field in the text sample description and uses the movie's background color instead. This flag's value is 0x0008.
Scroll in	Controls text scrolling. If this flag is set to 1, the text media handler scrolls the text in until the last of the text is in view. This flag's value is 0x0020.
Scroll out	Controls text scrolling. If this flag is set to 1, the text media handler scrolls the text out until the last of the text is gone. This flag's value is 0x0040.
Horizontal scroll	Controls text scrolling. If this flag is set to 1, the text media handler scrolls the text horizontally; otherwise, it scrolls the text vertically. This flag's value is 0x0080.
Reverse scroll	Controls text scrolling. If this flag is set to 1, the text media handler scrolls down (if scrolling vertically) or backward (if scrolling horizontally; horizontal scrolling also depends upon text justification). This flag's value is 0x0100.

Continuous scroll	Controls text scrolling. If this flag is set to 1, the text media handler displays new samples by scrolling out the old ones. This flag's value is 0x0200.
Drop shadow	Controls drop shadow. If this flag is set to1, the text media handler displays the text with a drop shadow. This flag's value is 0x1000.
Anti-alias	Controls anti-aliasing. If this flag is set to 1, the text media handler uses anti-aliasing when drawing text. This flag's value is 0x2000.
Key text	Controls background color. If this flag is set to 1, the text media handler does not display the background color, so that the text overlay background tracks. This flag's value is 0x4000.

Text justification	A 32-bit integer that indicates how the text should be aligned. Set this field to 0 for left-justified text, to 1 for centered text, and to –1 for right-justified text.
Background color	A 48-bit RGB color that specifies the text's background color.
Default text box	A 64-bit rectangle that specifies an area to receive text (top, left, bottom, right). Typically this field is set to all zeros.
Reserved	A 64-bit value that must be set to 0.
Font number	A 16-bit value that must be set to 0.
Font face	A 16-bit integer that indicates the font's style. Set this field to 0 for normal text. You can enable other style options by using one or more of the following bit masks:

0x0001	Bold
0x0002	Italic
0x0004	Underline
0x0008	Outline
0x0010	Shadow
0x0020	Condense
0x0040	Extend

Reserved	An 8-bit value that must be set to 0.
Reserved	A 16-bit value that must be set to 0.
Foreground color	A 48-bit RGB color that specifies the text's foreground color.
Text name	A Pascal string specifying the name of the font to use to display the text.

Text Sample Data

The format of the text data is a 16-bit length followed by the actual text. The length word specifies the number of bytes of text, not including the length word itself. Following the text, there may be one or more atoms containing additional information for drawing and searching the text.

Table 1-9 lists the currently defined text sample extensions.

Table 1-9 Text sample extensions

Text sample extension	Description
`'styl'`	Style information for the text. Allows you to override the default style in the sample description or to define more than one style for a sample. The data is a TextEdit style scrap.
`'ftab'`	Table of font names. Each table entry contains a font number (stored in a 16-bit integer) and a font name (stored in a Pascal string). This atom is required if the `'styl'` atom is present.
`'hlit'`	Highlight information. The atom data consists of two 32-bit integers. The first contains the starting offset for the highlighted text, and the second has the ending offset. Highlight samples can be in key frames or in differenced frames. When it's used in a differenced frame, the sample should contain a zero-length piece of text.
`'hclr'`	Highlight color. This atom specifies the 48-bit RGB color to use for highlighting.
`'drpo'`	Drop shadow offset. When the Display flags indicate drop shadow style, this atom can be used to override the default drop shadow placement. The data consists of two 16-bit integers. The first indicates the horizontal displacement of the drop shadow, in pixels; the second, the vertical displacement.
`'drpt'`	Drop shadow transparency. The data is a 16-bit integer between 0 and 256 indicating the degree of transparency of the drop shadow. A value of 256 makes the drop shadow completely opaque.
`'imag'`	Image font data. This atom contains two more atoms. An `'idat'` atom contains compressed image data used to draw the text when the required fonts are not available. An `'idsc'` atom contains a video sample description describing the format of the compressed image data.
`'metr'`	Image font highlighting. This atom contains metric information that governs highlighting when an `'imag'` atom is used for drawing.

Music Media

Music media is used to store note-based audio data, such as MIDI data, in QuickTime movies. It has a media type of `'musi'`.

Music Sample Description

The music sample description uses the standard sample description header, as described in "Sample Table Atoms" beginning on page 45.

The data format field in the sample description is always set to `'musi'`. The music media handler adds an additional 32-bit integer field to the sample description containing flags. Currently no flags are defined, and this field should be set to 0.

Following the Flags field, there may be appended data in the QuickTime music format. This data consists of part-to-instrument mappings in the form of general events containing note requests. One note request event should be present for each part that will be used in the sample data.

Music Sample Data

The sample data for music samples consists entirely of data in the QuickTime music format. Typically, up to 30 seconds of notes will be grouped into a single sample. For a complete description of the QuickTime music format, see your most recent QuickTime developer update documentation.

MPEG Media

MPEG media is used to store MPEG streams in QuickTime movies. It has a media type of `'MPEG'`.

MPEG Sample Description

The MPEG sample description uses the standard sample description header, as described in “Sample Table Atoms” beginning on page 45.

The Data format field in the sample description is always set to `'MPEG'`. The MPEG media handler adds no additional fields to the sample description.

MPEG Sample Data

Each sample in an MPEG media is an entire MPEG stream. This can mean that a single MPEG sample may be several hundred megabytes in size. The MPEG encoding used by QuickTime corresponds to the ISO standard, as described in ISO document CD 11172.

Sprite Media

Sprite media is used to store character-based animation data in QuickTime movies. It has a media type of `'sprt'`.

Sprite Sample Description

The base sample description uses the standard sample description header, as described in “Sample Table Atoms” beginning on page 45.

The Data format field in the sample description is always set to `'sprt'`. The base media handler adds no additional fields to the sample description.

Sprite Sample Data

All sprite samples are stored in QT atom structures. The sprite media uses both key frames and differenced frames. The key frames contain all of the sprite's image data and the initial settings for each of the sprite's properties.

A key frame always contains a shared data atom of type `'dflt'`. This atom contains data to be shared between the sprites, consisting mainly of image data and sample descriptions. The shared data atom contains a single sprite image container atom, with an atom type value of `'imct'` and an ID value of 1.

The sprite image container atom stores one or more sprite image atoms of type `'imag'`. Each sprite image atom contains a video sample description immediately followed by the sprite's compressed image data. The sprite image atoms should have ID numbers starting at 1 and counting consecutively upward.

The key frame must also contain definitions for each sprite in atoms of type `'sprt'`. Sprite atoms should have ID numbers that start at 1 and count consecutively upward. Each sprite atom contains a list of properties. Four of the properties are required: Image index, Matrix, Layer, and Visibility. Table 1-10 shows all currently defined sprite properties.

Table 1-10 Sprite properties

Property Name	Property Value	Description
Matrix	1	Specifies a transformation matrix that describes the sprite's location, scaling, and so on.
Layer	5	Contains a 16-bit integer value specifying the layer into which the sprite is to be drawn (lower layers are rendered on top of higher layers)
Visibility	4	Contains a 16-bit integer that controls the sprite's visibility: set to 1 if the sprite is visible; 0 if not
Graphics mode	6	Contains a 32-bit integer specifying the sprite's graphics mode; this value is always followed by a 48-bit RGB value
Image index	100	Contains the atom ID of the sprite's image atom

The frame difference sample differs from the key frame sample in two ways. First, the frame difference sample does not contain a shared data atom. All shared data must appear in the key frame. Second, only those sprite properties that change need to be specified. If none of a sprite's properties change in a given frame, then the sprite does not need an atom in the differenced frame.

The frame difference sample can be used in one of two ways: the frame differences can be combined, as with video key frames, to construct the current frame; or the current frame can be derived by combining only the key frame and the current frame difference.

Base Media

Base media is used to store data that is not intended to be displayed in QuickTime movies. The data may be stored for an application to retrieve, or it may be used to modify another track. It has a media type of `'gnrc'`. Other media types are often built on top of the base media. Examples of media types derived from the base media include MPEG, sprite, text, music, and time code.

Base Sample Description

The base sample description uses the standard sample description header, as described in "Sample Table Atoms" beginning on page 45.

The Data format field in the sample description is always set to `'gnrc'`. The base media handler adds no additional fields to the sample description.

Base Sample Data

The base media handler defines no data formats. Applications may put whatever data they require into the samples.

Tween Media

Tween media is used to store pairs of values to be interpolated between in QuickTime movies. These interpolated values modify the playback of other media types by using track references and track input maps. For example, a tween media can generate gradually-changing relative volume levels to cause an audio track to fade out. It has a media type of `'twen'`.

Tween Sample Description

The tween sample description uses the standard sample description header, as described in "Sample Table Atoms" beginning on page 45.

The Data format field in the sample description is always set to `'twen'`. The tween media handler adds no additional fields to the sample description.

Tween Sample Data

Tween sample data is stored in QT atom structures.

At the root level, there are one or more tween entry atoms; these atoms have an atom type value of `'twen'`. Each tween entry atom completely describes one interpolation operation. These atoms should be consecutively numbered starting at 1, using the Atom ID field.

Each tween entry atom contains several more atoms that describe how to perform the interpolation. The Atom ID field in each of these atoms must be set to 1.

- Tween start atom (atom type is `'twst'`). This atom specifies the time at which the interpolation is to start. The time is expressed in the media's time coordinate system, relative to the start of the media sample. If this atom is not present, the starting offset is assumed to be 0.

- Tween duration atom (atom type is `'twdu'`). This atom specifies how long the interpolation is to last. The time is expressed in the media's time coordinate system. If this atom is not present, the duration is assumed to be the length of the sample.
- Tween data atom (atom type is `'twdt'`). This atom contains the actual values for the interpolation. The contents depend on the value of the tween type atom.
- Tween type atom (atom type is `'twnt'`). Describes the type of interpolation to perform. Table 1-11 shows all currently defined tween types. All tween types are currently supported using linear interpolation.

Table 1-11 Tween type values

Tween type	Value	Tween data
16-bit integer	1	Two 16-bit integers.
32-bit integer	2	Two 32-bit integers.
32-bit fixed-point	3	Two 32-bit fixed-point numbers.
Point: two 16-bit integers	4	Two points.
Rectangle: four 16-bit integers	5	Two rectangles.
QuickDraw region	6	Two matrices. The tween entry atom must contain a `'qdrg'` atom with an Atom ID value of 1. The region is transformed through the interpolated matrices.
Matrix	7	Two matrices.
RGB color: three 16-bit integers	8	Two RGB colors.
Graphics mode with RGB color	9	Two graphics modes with RGB color. Only the RGB color is interpolated. The graphics modes must be the same.

3D Media

QuickTime movies store 3D image data in a base media. This media has a media type of `'qd3d'`.

3D Sample Description

The 3D sample description uses the standard sample description header, as described in "Sample Table Atoms" beginning on page 45.

The Data format field in the sample description is always set to `'qd3d'`. The 3D media handler adds no additional fields to the sample description.

3D Sample Data

The 3D samples are stored in the 3D metafile format developed for QuickDraw 3D. Please refer to *Inside Macintosh: 3D Metafile Reference* for details.

Basic Data Types

This section describes a number of common data types that are used in QuickTime files. For information about other QuickTime data types, refer to *Inside Macintosh: QuickTime*.

Language Code Values

Some elements of a QuickTime file may be associated with a particular spoken language. To indicate the language associated with a particular object, the QuickTime file format uses language codes from the Macintosh Script Manager. Table 1-12 lists the language codes supported by QuickTime.

Table 1-12 QuickTime language code values

Language	Value	Language	Value
English	0	Georgian	52
French	1	Moldavian	53
German	2	Kirghiz	54
Italian	3	Tajiki	55
Dutch	4	Turkmen	56
Swedish	5	Mongolian	57
Spanish	6	MongolianCyr	58
Danish	7	Pashto	59
Portuguese	8	Kurdish	60
Norwegian	9	Kashmiri	61
Hebrew	10	Sindhi	62
Japanese	11	Tibetan	63
Arabic	12	Nepali	64
Finnish	13	Sanskrit	65
Greek	14	Marathi	66
Icelandic	15	Bengali	67
Maltese	16	Assamese	68

Table 1-12 QuickTime language code values (continued)

Language	Value	Language	Value
Turkish	17	Gujarati	69
Croatian	18	Punjabi	70
Traditional Chinese	19	Oriya	71
Urdu	20	Malayalam	72
Hindi	21	Kannada	73
Thai	22	Tamil	74
Korean	23	Telugu	75
Lithuanian	24	Sinhalese	76
Polish	25	Burmese	77
Hungarian	26	Khmer	78
Estonian	27	Lao	79
Lettish	28	Vietnamese	80
Latvian	28	Indonesian	81
Saamisk	29	Tagalog	82
Lappish	29	MalayRoman	83
Faeroese	30	MalayArabic	84
Farsi	31	Amharic	85
Persian	31	Tigrinya	86
Russian	32	Galla	87
Simplified Chinese	33	Oromo	87
Flemish	34	Somali	88
Irish	35	Swahili	89
Albanian	36	Ruanda	90
Romanian	37	Rundi	91
Czech	38	Chewa	92
Slovak	39	Malagasy	93
Slovenian	40	Esperanto	94
Yiddish	41	Welsh	128
Serbian	42	Basque	129
Macedonian	43	Catalan	130
Bulgarian	44	Latin	131
Ukrainian	45	Quechua	132

Table 1-12 QuickTime language code values (continued)

Language	Value	Language	Value
Byelorussian	46	Guarani	133
Uzbek	47	Aymara	134
Kazakh	48	Tatar	135
Azerbaijani	49	Uighur	136
AzerbaijanAr	50	Dzongkha	137
Armenian	51	JavaneseRom	138
Georgian	52		

Calendar Date and Time Values

QuickTime movies store date and time information in Macintosh date format: a 32-bit value indicating the number of seconds that have passed since midnight, January 1, 1904.

Matrices

QuickTime files use matrices to describe spatial information about many objects, such as tracks within a movie.

A transformation matrix defines how to map points from one coordinate space into another coordinate space. By modifying the contents of a transformation matrix, you can perform several standard graphics display operations, including translation, rotation, and scaling. The matrix used to accomplish two-dimensional transformations is described mathematically by a 3-by-3 matrix.

All values in the matrix are 32-bit fixed-point numbers divided as 16.16, except for the {u, v, w} column that contains 32-bit fixed-point numbers divided as 2.30. Figure 1-44 depicts how QuickTime uses matrices to transform displayed objects.

Figure 1-44 How display matrices are used in QuickTime

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} a & b & u \\ c & d & v \\ t_x & t_y & w \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

Graphics Modes

QuickTime files use graphics modes to describe how one video or graphics layer should be combined with the layers beneath it. Graphics modes are also known as transfer modes. Some graphics modes require a color to be specified for certain operations, such as blending to determine the blend level. QuickTime uses the graphics modes defined by QuickDraw. For information on QuickDraw graphics modes, see *Inside Macintosh: Imaging*.

The most common graphics modes are `srcCopy` (0x00) and `ditherCopy` (0x040), which simply indicate that the image should not blend with the image behind it, but overwrite it. QuickTime also defines several new graphics modes.

Table 1-13 lists the new graphics modes supported by QuickTime.

Table 1-13 QuickTime graphics modes

Graphics mode value	Description
0x0100	Straight alpha
0x0101	Alpha pre-multiplied with white
0x0102	Alpha pre-multiplied with black

RGB Colors

Many atoms in the QuickTime file format contain RGB color values. These are usually stored as three consecutive unsigned 16-bit integers in the following order: red, green, blue.

Examples

This section contains a number of examples that help you pull together all of the material in this book by examining the atom structure that results from a number of different scenarios. This section is divided into the following topics:

- "Creating Video Tracks at 30 Frames-per-second" discusses creating 30 fps video
- "Creating Video Tracks at 29.97 Frames-per-second" describes creating 29.97 fps video
- "Creating Audio Tracks at 44.1Khz" provides an example of creating an audio track
- "Creating a Time Code Track for 29.97 FPS Video" presents a time code track example
- "Playing With Edit Lists" discusses how to interpret edit list data
- "Interleaving Movie Data" shows how a movie's tracks are interleaved in the movie data file

- "Referencing Two Data Files With a Single Track" shows how track data may reside in more than one file

Creating Video Tracks at 30 Frames-per-second

The duration of a video frame is stored in the time-to-sample atom contained within a sample table atom. This duration cannot be interpreted without the media's time scale, which defines the units-per-second for the duration. In this example, each frame has the same duration, so the time-to-sample atom has one entry that applies to all video frames in the media.

As long as the ratio between frame duration and media time scale remains 1:30, any combination of values can be used for the duration and time scale. The larger the time scale the shorter the maximum duration. Since a movie defaults to a time scale of 600, this is a good number to use. It is also the least common multiple for 24, 25, and 30, making it handy for much of the math you are likely to encounter when making a movie.

The movie time scale is independent of the media time scale. Since you want to avoid movie edits that don't land on frame boundaries, it is a good idea to keep the movie time scale and the media time scale the same, or the movie time scale should be an even multiple of the media time scale. The movie time scale is stored in the movie header atom.

With a time scale of 600 in the media header atom, the time-to-sample atom would contain the following data values:

Atom size	24
Atom type	`'stts'`
Version/Flags	0
Number of entries	1
Sample count	n
Sample duration	20

Creating Video Tracks at 29.97 Frames-per-second

NTSC color video is not 30 frames-per-second (fps), but actually 29.97 fps. The previous example showed how the media time scale and the duration of the frames specify the video's frame rate. By setting the media's time scale to 2997 units per second and setting the frame durations to 100 units each, the effective rate is 29.97 fps exactly.

In this situation, it is also a good idea to set the movie time scale to 2997 in order to avoid movie edits that don't land on frame boundaries. The movie's time scale is stored in the movie header atom.

With a time scale of 2997 in the media header atom, the time-to-sample atom would contain the following data values:

Atom size	24
Atom type	`'stts'`
Version/Flags	0
Number of entries	1
Sample count	n
Sample duration	100

Creating Audio Tracks at 44.1Khz

The duration of an audio sample is stored in the time-to-sample atom contained in a sample table atom. This duration cannot be interpreted without the media's time scale, which defines the units-per-second for the duration. With audio, the duration of each audio sample is typically 1, so the time-to-sample atom has one entry that applies to all audio samples.

With a time scale of 44100 in the media header atom, the time-to-sample atom would contain the following data values:

Atom size	24
Atom type	`'stts'`
Version/Flags	0
Number of entries	1
Sample count	n
Sample duration	1

This atom does not indicate whether the audio is stereo or mono or whether it contains 8-bit or 16-bit samples. That information is stored in the sound sample description atom, which is contained in the sample table atom.

Creating a Time Code Track for 29.97 FPS Video

A time code track specifies time code information for other tracks. The time code keeps track of the time codes of the original source of the video and audio. After a movie has been edited, the time code can be extracted to determine the source tape and the time codes of the frames.

It is important that the time code track has the same time scale as the video track. Otherwise, the time code will not tick at the exact same time as the video track.

For each contiguous source tape segment, there is a single time code sample that specifies the time code value corresponding to the start of the segment. From this sample, the time code value can be determined for any point in the segment.

The sample description for a time code track specifies the time code system being used (for example, 30 fps drop-frame) and the source information. Each sample is a time code value.

Since the time code media handler is a derived from the base media handler, the media information atom starts with a generic media header atom. The time code atoms would contain the following data values:

Atom size	77				
Atom type	`'gmhd'`				
		Atom size	69		
		Atom type	`'gmin'`		
		Version/Flags	0		
		Graphics mode	0x0040		
		Opcolor (red)	0x8000		
		Opcolor (green)	0x8000		
		Opcolor (blue)	0x8000		
		Balance	0		
		Reserved	0		
		Atom size	45		
		Atom type	`'tmcd'`		
				Atom size	37
				Atom type	`'tcmi'`
				Version/Flags	0
				Text font	0 (system font)
				Text face	0 (plain)
				Text size	12
				Text color (red)	0
				Text color (green)	0
				Text color (blue)	0
				Background color (red)	0
				Background color (green)	0
				Background color (blue)	0
				Font name	`"\pChicago"` (Pascal string)

The sample table atom contains all the standard sample atoms and has the following data values:

Atom size	174		
Atom type	'stbl' (sample table)		
	Atom size	74	
	Atom type	'stsd' (sample description)	
	Version/Flags	0	
	Number of entries	1	
	Sample description size [1]	58	
	Data format [1]	'tmcd'	
	Reserved [1]	0	
	Data reference index [1]	1	
	Flags [1]	0	
	Flags (time code) [1]	7 (drop frame + 24 hour + negative times ok)	
	Time scale[1]	2997	
	Frame duration[1]	100	
	Number of frames[1]	20	
		Atom size	24
		Atom type	'name'
		String length	12
		Language code	0 (English)
		Name	“my tape name”
	Atom size	24	
	Atom type	'stts' (time-to-sample)	
	Version/Flags	0	
	Number of entries	1	
	Sample count[1]	1	
	Sample duration[1]	1	
	Atom size	28	
	Atom type	'stsc' (sample-to-chunk)	
	Version/Flags	0	
	Number of entries	1	
	First chunk[1]	1	
	Samples per chunk[1]	1	

Sample description ID[1]	1
Atom size	20
Atom type	`'stsz'` (sample size)
Version/Flags	0
Sample size	4
Number of entries	1
Atom size	20
Atom type	`'stco'` (chunk offset)
Version/Flags	0
Number of entries	1
Offset[1]	(offset into file of chunk 1)

In the example, let's assume that the segment's beginning time code is 1:15:32.4 (1 hour, 15 minutes, 32 seconds, and 4 frames). This time would be expressed in the data file as 0x010F2004 (0x01 = 1 hour; 0x0F = 15 minutes; 0x20 = 32 seconds; 0x04 = 4 frames).

The video and audio tracks must contain a track reference atom to indicate that they reference this time code track. The track reference is the same for both and is contained in the track atom (at the same level as the track header and media atoms).

This track reference would contain the following data values:

Atom size	12
Atom type	`'tref'`
Reference type	`'tmcd'`
Track ID of referenced track (time code track)	3

In this example, the video and sound tracks are tracks 1 and 2. The time code track is track 3.

Playing With Edit Lists

A segment of a movie can be repeated without duplicating media data by using edit lists. Suppose you have a single-track movie whose media time scale is 100 and track duration is 1000 (10 seconds). For this example the movie's time scale is 600. If there are no edits in the movie, the edit atom would contain the following data values:

Atom size	36	
Atom type	`'edts'`	
	Atom size	28
	Atom type	`'elst'`
	Version/Flags	0
	Number of entries	1
	Track duration	6000 (10 seconds)
	Media time	0
	Media rate	1.0

Because this is a single-track move, the track's duration in the track header atom is 6000, and the movie's duration in the movie header atom is 6000.

If you change the track to play the media from time 0 to time 2 seconds, and then play the media from time 0 to time 10 seconds, the edit atom would now contain these data values:

Atom size	48	
Atom type	`'edts'`	
	Atom size	40
	Atom type	`'elst'`
	Version/Flags	0
	Number of entries	2
	Track duration[1]	1200 (2 seconds)
	Media time[1]	0
	Media rate[1]	1.0
	Track duration[2]	6000 (10 seconds)
	Media time[2]	0
	Media rate[2]	1.0

Because the track is now 2 seconds longer, the track's duration in the track header atom must now be 7200, and the movie's duration in the movie header atom must also be 7200.

Currently, the media plays from time 0 to time 2, then plays from time 0 to time 10. If you take that repeated segment at the beginning (time 0 to time 2) and play it at double speed to maintain the original duration, the edit atom would now contain the following values:

Atom size	60	
Atom type	`'edts'`	
	Atom size	52
	Atom type	`'elst'`
	Version/Flags	0
	Number of entries	3
	Track duration[1]	600 (1 seconds)
	Media time[1]	0
	Media rate[1]	2.0
	Track duration[2]	600 (1 second)
	Media time[2]	0
	Media rate[2]	2.0
	Track duration[3]	4800 (8 seconds)
	Media time[3]	200
	Media rate[3]	1.0

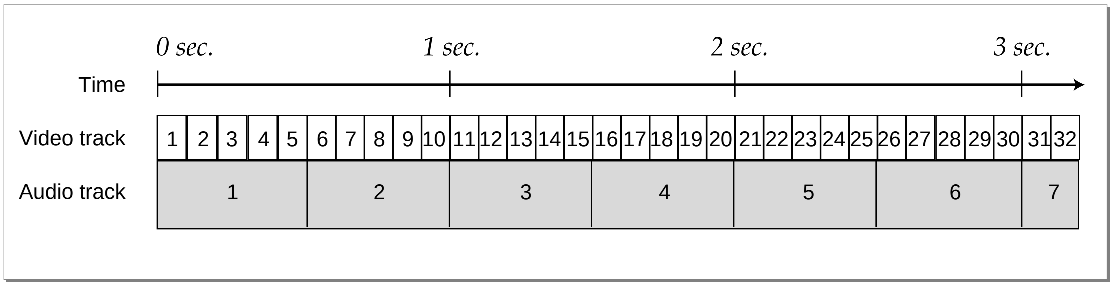
Because the track is now back to its original duration of 10 seconds, its duration in the track header atom is 6000 and the movie’s duration in the movie header atom is 6000.

Interleaving Movie Data

In order to get optimal movie playback, you must create the movie with interleaved data. The data for the movie is placed on disk in time order so the video, sound, and other data for a particular time in the movie are close together in the file. This means that you will have to intersperse the data from different tracks. To illustrate this, consider a movie with a single video and a single audio track.

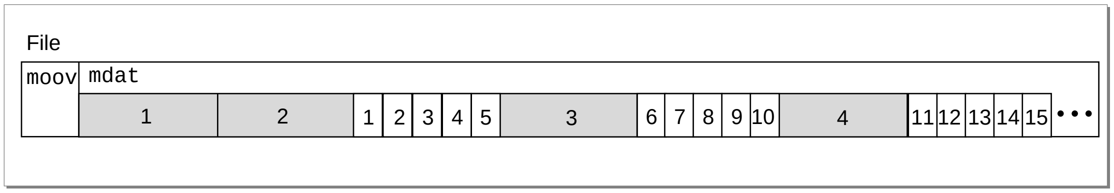
Figure 1-45 shows how the movie data was collected, and how the data would need to be played back for proper synchronization. In this example, the video data is recorded at 10 frames per second and the audio data is grouped into 1/2-second chunks.

Figure 1-45 Noninterleaved movie data



After the data has been interleaved on the disk, the movie data atom contains movie data in the order shown in Figure 1-46.

Figure 1-46 Interleaved movie data



In this example, the file begins with the movie atom (`'moov'`), followed by the movie data atom (`'mdat'`). In order to overcome any latencies in sound playback, at least one second of sound data is placed at the beginning of the interleaved data. This means that the sound and video data are offset from each other in the file by one second.

Referencing Two Data Files With a Single Track

The data reference index to be used for a given media sample is stored within that sample's sample description. Therefore, a track must contain multiple sample descriptions in order for that track to reference multiple data files. A different sample description must be used whenever the data file changes or whenever the format of the data changes. The sample-to-chunk atom determines which sample description to use for a sample.

The sample description atom would contain the following data values:

Atom size	…		
Atom type	`'stsd'`		
Version/Flags	0		
Number of entries	2		
	Sample description size[1]	…	
	Data format	`'tmcd'`	
	Reserved	0	
	Data reference index	1	
	(sample data)	…	
	Sample description size[1]	…	
	Data format	`'tmcd'`	
	Reserved	0	
	Data reference index	2	
	(sample data)	…	

If there were only 1 sample per chunk and the first 10 samples were extracted from sample description 2 and the next 30 samples were extracted from sample description 1, the sample-to-chunk atom would contain the following data values:

Atom size	40
Atom type	`'stsc'`
Version/Flags	0
Number of entries	2
First chunk[1]	1
Samples per chunk[1]	1
Sample description ID[1]	2
First chunk[2]	11
Samples per chunk[2]	1
Sample description ID[2]	1

The data reference atom would contain the following data values:

Atom size	…	
Atom type	`'dinf'`	
	Atom size	…
	Atom type	`'dref'`
	Version/Flags	0
	Number of entries	2
	Size[1]	…
	Type[1]	`'alis'`
	Version[1]	0
	Flags[1]	0 (not self referenced)
	Data reference[1]	[alias pointing to file #1]
	Size[2]	…
	Type[2]	`'alis'`
	Version[2]	0
	Flags[2]	0 (not self referenced)
	Data reference[2]	[alias pointing to file #2]

Index

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages and final pages were created on an Apple LaserWriter Pro printer. Line art was created using Adobe™ Illustrator and Adobe Photoshop. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

LEAD WRITER
Linda Kyrnitszke

WRITER
Doug Engfer

DEVELOPMENTAL EDITOR
Wendy Krafft

ILLUSTRATORS
Deb Dennis, Sandee Karr

PRODUCTION EDITOR
Gerri Gray

Special thanks to Peter Hoddie

OpenOffice.org
Source Project

OpenOffice.org's Documentation of the

Microsoft® Excel File Format

Excel Versions 2, 3, 4, 5, 95, 97, 2000, XP

Author	Daniel Rentz <daniel.rentz@sun.com>
Source	PDF: http://sc.openoffice.org/excelfileformat.pdf XML: http://sc.openoffice.org/excelfileformat.sxw
Project started	2001-Jun-29
Last change	2001-Nov-24

Contents

1 Introduction

1.1 File Format Versions

The Excel file format is named BIFF (Binary Interchange File Format). The following table shows which Excel version writes which file format.

Excel version	BIFF version	Document type	File format
Excel 2	BIFF2	Worksheet	Stream
Excel 3	BIFF3	Worksheet	Stream
Excel 4	BIFF4	Worksheet or workbook	Stream
Excel 5.0	BIFF5	Workbook	OLE2 storage
Excel 7.0 (Excel 95)	BIFF7	Workbook	OLE2 storage
Excel 97, 2000, XP	BIFF8	Workbook	OLE2 storage

The oldest file format BIFF2 has of course the most restrictions. From BIFF4 on it is possible to store a bundle of sheets, called a workbook. The current format BIFF8 contains major changes towards older BIFF versions, for instance the handling of Unicode strings.

1.2 Structure of a Worksheet File (BIFF2-BIFF4)

Files stored in the BIFF versions BIFF2 to BIFF4 contain all records for a sheet or a BIFF4 workbook in one simple stream. The records are arranged sequential, they are never embedded in other records.

1.3 Structure of a Workbook File (BIFF5-BIFF8)

An Excel workbook with several sheets (from BIFF5 on) is stored as an OLE2 compound file. It contains several streams for different types of data. The following table lists names of possible streams.

Stream name	Contents
Book	BIFF5/BIFF7 workbook stream (→4.3)
Workbook	BIFF8 workbook stream (→4.3)
<05_H>SummaryInformation	Document settings
<05_H>DocumentSummaryInformation	Document settings
User Names	User names in shared workbooks (→9)
Revision Log	Change tracking log stream (→9)

The names of the streams SummaryInformation and DocumentSummaryInformation contain a leading byte with the value 05_H.

It is possible to create substorages like subdirectories in a file system, for instance for the pivot table streams. These storages contain substreams itself.

Storage name	Contents
LNKxxxxxxxx	Storage for a linked OLE object (→6)
MBDxxxxxxxx	Storage for an embedded OLE object (→6)
_SX_DB_CUR	Pivot cache storage. The streams contain cached values for one or more PivotTables (→8).
_VBA_PROJECT_CUR	Visual BASIC project storage

In all streams the records are arranged sequential, they are never embedded in other records. Exception in BIFF8: The Escher object stream is splitted and embedded in several MSODRAWING records (→6).

1.4 Structure of a Record

In an Excel data stream the data is divided into several records. Each record contains specific data for the various features of Excel. The common structure of a record is described in the following table.

Offset	Size	Contents	
0	2	Identifier	} Record header
2	2	Size of the following data (sz)	
4	sz	Data	

The maximum size of the record data is limited. If the size of the record data exceeds the given limits, one or more CONTINUE (→5.6) records will be added. Inside of a CONTINUE record the data of the previous record continues as usual.

In the following descriptions only the record data without the headers is shown. All offsets are relative to the beginning of the record data and not to the entire record. The contents of most of the records differ from version to version. This will be described in separate tables. A few older records are replaced in newer BIFF versions. Excel does not write these old records anymore, but can still read them.

1.5 Byte Order

All data items containing more than one byte are stored using the Little-Endian method. That means the least significant byte is stored first and the most significant byte last. This is valid for all data types like 16-bit-integers, 32-bit-integers, floating-point values and Unicode characters. For instance the 16-bit-integer value 1234_H is converted into the byte sequence 34_H 12_H.

2 Basic Substructures

This chapter contains information about substructures which are part of several records, for instance strings or error codes.

2.1 Byte Strings (BIFF2-BIFF7)

All Excel file formats up to BIFF7 contain simple byte strings. The byte string consists of the length of the string followed by the character array. The length is stored either as 8-bit-integer or as 16-bit-integer, depending on the current record. The string is not zero-terminated.

Offset	Size	Contents
0	1 or 2	Length of the string (character count) (`ln`)
1 or 2	`ln`	Character array (8-bit-characters)

2.2 Unicode Strings (BIFF8)

From BIFF8 on, strings are stored in a new Unicode format which allows reading and writing 16-bit-characters. The following tables describe the standard format, but in many records the strings differ from this format. This will be mentioned separately. It is possible (but not required) to store Rich-Text formatting information and extended information for Far-East inside of an Unicode string. This results in four different ways to store a string. The string is not zero-terminated.

2.2.1 Contents of an Unicode string

The string consists of the character count (as usual an 8-bit-integer or a 16-bit-integer), option flags, the character array and optional formatting information. If the string is empty, sometimes the option flags field will not occur. This is mentioned at the respective place.

• Unicode string without additional information

Offset	Size	Contents
0	1 or 2	Length of the string (character count) (`ln`)
1 or 2	1	Option flags (see below): 00_H or 01_H
2 or 3	`ln` or 2·`ln`	Character array (8-bit-characters or 16-bit-characters)

• Unicode string with Rich-Text formatting information

Offset	Size	Contents
0	1 or 2	Length of the string (character count) (`ln`)
1 or 2	1	Option flags (see below): 08_H or 09_H
2 or 3	2	Number of Rich-Text formatting runs (`rt`)
4 or 5	`ln` or 2·`ln`	Character array (8-bit-characters or 16-bit-characters)
var.	4·`rt`	List of `rt` formatting runs. Each run contains two 16-bit indexes:

Offset	Size	Contents
0	2	First formatted character (zero-based)
2	2	Index to FONT record (➜5.15)

- **Unicode string with Far-East information**

Offset	Size	Contents
0	1 or 2	Length of the string (character count) (ln)
1 or 2	1	Option flags (see below): 04_H or 05_H
2 or 3	4	Far-East data size (sz)
6 or 7	ln or 2·ln	Character array (8-bit-characters or 16-bit-characters)
var.	sz	Unknown extended data about phonetic, keyboard, etc.

- **Unicode string with Rich-Text and Far-East information**

Offset	Size	Contents
0	1 or 2	Length of the string (character count) (ln)
1 or 2	1	Option flags (see below): $0C_H$ or $0D_H$
2 or 3	2	Number of Rich-Text formatting runs (rt)
4 or 5	4	Far-East data size (sz)
8 or 9	ln or 2·ln	Character array (8-bit-characters or 16-bit-characters)
var.	4·rt	List of rt formatting runs. See above for details.
var.	sz	Unknown extended data about phonetic, keyboard, etc.

2.2.2 Option flags

Bit	Mask	Contents	
0	01_H	0 = 8-bit-characters	1 = 16-bit-characters
2	04_H	0 = Contains no Far-East info	1 = Contains Far-East info
3	08_H	0 = Contains no Rich-Text info	1 = Contains Rich-Text info

2.3 RK Values

An RK value is an encoded integer or floating-point value. RK values have a size of 4 bytes and are used to decrease file size for floating-point values.
Structure of an RK value (32-bit-value):

Bit	Mask	Contents	
0	00000001_H	0 = Value not changed	1 = Value multiplied by 100
1	00000002_H	0 = IEEE floating-point value	1 = Integer value
31-2	$FFFFFFFC_H$	Encoded value	

If bit 1 is set to 0, the encoded value represents the 30 most significant bits of an IEEE floating-point value. The 34 least significant bits must be set to zero. If bit 1 is set to 1, the encoded value represents a signed 30-bit-integer value.
If bit 0 is set to 1, the decoded value must be divided by 100 to get the final result.
Examples:

RK value	Decoded value	Result
$3FF00000_H$	Floating-point = 1	1
$3FF00001_H$	Floating-point = 1	0.01
$004B5646_H$	Integer = 1234321	1234321
$004B5647_H$	Integer = 1234321	12343.21

2.4 Error Codes

If the calculation of a formula results in an error or any other action fails, Excel sets a specific error code. These error codes are used for instance in cell records and formulas.

Error code	Error value	Description
00_H	#NULL!	Intersection of two cell ranges is empty
07_H	#DIV/0!	Division by zero
$0F_H$	#VALUE!	Wrong type of operand
17_H	#REF!	Illegal or deleted cell reference
$1D_H$	#NAME?	Wrong function or range name
24_H	#NUM!	Value range overflow
$2A_H$	#N/A!	Argument or function not available

2.5 List of Cached Values

The records CRN (→5.7) and EXTERNNAME (→5.12) and the formula token ptgArray (array constant, →3.10.1) require a list of constant values (floating-point values, strings, boolean values or error codes). These values are stored as a simple list. The number of values is stored before in the respective record or token.

- **IEEE floating-point value**

Offset	Size	Contents
0	1	01_H (identifier for a floating-point constant)
1	8	IEEE floating-point value

- **String value**

A string value, BIFF2-BIFF7:

Offset	Size	Contents
0	1	02_H (identifier for a string constant)
1	var.	Byte string, 8-bit string length (→2.1)

A string value, BIFF8:

Offset	Size	Contents
0	1	02_H (identifier for a string constant)
1	var.	Unicode string, 16-bit string length, option flags occur always (→2.2)

- **Boolean value**

Offset	Size	Contents
0	1	04_H (identifier for a boolean constant)
1	1	0 = FALSE, 1 = TRUE
2	7	Not used

- **Error value**

Offset	Size	Contents
0	1	10_H (identifier for an error constant)
1	1	Error code (→2.4)
2	7	Not used

2.6 Encoded Document Names

2.6.1 Encoded file names

The intention of encoding file names is to make them more platform independent. Encoded file names occur in the records EXTERNSHEET (BIFF2-BIFF7, →5.13) or SUPBOOK (BIFF8, →5.35) and DCONREF (→5.8).

The first character of the file name is used to determine the type of encoding. In Unicode strings (BIFF8) this could be a 16-bit-value.

First character	Meaning
00_H	Empty sheet name (nothing will follow)
01_H	Encoded file name
02_H	External reference to the own document (nothing will follow)
03_H	External reference to a sheet in the own document (BIFF5/BIFF7)
others	Not encoded. This is the first character of the file name.

Inside of the encoded file name there can occur several control characters.

Control character	Meaning
01_H	An MS-DOS drive letter will follow or „@“ for a local network path
02_H	Start path name on same drive as own document
03_H	End of subdirectory name
04_H	Start path name in parent directory of own document (may occur repeatedly)
06_H	Start path name in startup directory of Excel
09_H	Sheet in the same workbook (BIFF4)

Example: Own document is saved as „C:\path\own.xls“.

Formula	Encoded filename
`=own.xls!A1`	<02_H>
`=Sheet2!A1`	<01_H><09_H>Sheet2 (BIFF4 workbook)
`=Sheet2!A1`	<03_H>Sheet2 (BIFF5/BIFF7)
`=[ext.xls]Sheet1!A1`	<01_H>[ext.xls]Sheet1
`='sub\[ext.xls]'Sheet1!A1`	<01_H>sub<03_H>[ext.xls]Sheet1
`='\[ext.xls]'Sheet1!A1`	<01_H><02_H>[ext.xls]Sheet1
`='\sub\[ext.xls]'Sheet1!A1`	<01_H><02_H>sub<03_H>[ext.xls]Sheet1
`='\sub\sub2\[ext.xls]'Sheet1!A1`	<01_H><02_H>sub<03_H>sub2<03_H>[ext.xls]Sheet1
`='D:\sub\[ext.xls]'Sheet1!A1`	<01_H><01_H>Dsub<03_H>[ext.xls]Sheet1
`='..\sub\[ext.xls]'Sheet1!A1`	<01_H><04_H>sub<03_H>[ext.xls]Sheet1
`='\\pc\sub\[ext.xls]'Sheet1!A1`	<01_H><01_H>@pc<03_H>sub<03_H>[ext.xls]Sheet1

2.6.2 Encoded document names for DDE and OLE object links

A DDE link contains the name of the server application and the name of a document. An OLE object link contains a class name and a document name. In both cases the names are stored in one string, separated by the control character 03_H.

Example: A document contains a DDE link to the SO/OOo Calc document „example.sxc“ and an OLE object link to the bitmap file „example.bmp“.

Link	Encoded document name
DDE	soffice<03_H>example.sxc
OLE object	Package<03_H>example.bmp

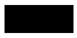
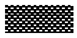
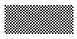
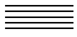
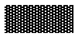
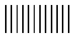
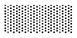
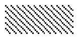
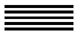
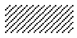
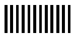
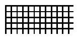
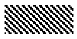
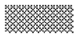
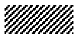
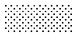
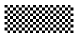
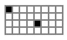
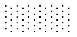
2.7 Line Styles for Cell Borders (BIFF3-BIFF8)

These line styles are used to define cell borders. The styles `08`$_H$ to `0D`$_H$ are available in BIFF8 only.

Index	Style	Sample	Index	Style	Sample
`00`$_H$	No line			The following for BIFF8 only:	
`01`$_H$	Thin	────────	`08`$_H$	Medium dashed	▬ ▬ ▬ ▬ ▬ ▬ ▬
`02`$_H$	Medium	━━━━━━━━	`09`$_H$	Thin dash-dotted	–·–·–·–·–·–
`03`$_H$	Dashed	– – – – – – – – –	`0A`$_H$	Medium dash-dotted	▬·▬·▬·▬·▬
`04`$_H$	Dotted	- - - - - - - - - - - -	`0B`$_H$	Thin dash-dot-dotted	–··–··–··–··–
`05`$_H$	Thick	████████	`0C`$_H$	Medium dash-dot-dotted	▬··▬··▬··▬
`06`$_H$	Double	════════	`0D`$_H$	Slanted medium dash-dotted	▬⸝▬⸝▬⸝▬⸝▬
`07`$_H$	Hair	⋯⋯⋯⋯⋯⋯			

2.8 Patterns for Cell Background Area (BIFF3-BIFF8)

From BIFF3 on, the cell background area may contain a pattern. Foreground and background colors of the pattern are defined separately. In the following table black is used as foreground color and white as background color.

Index	Pattern	Sample	Index	Pattern	Sample
`00`$_H$		No background			
`01`$_H$			`0A`$_H$		
`02`$_H$			`0B`$_H$		
`03`$_H$			`0C`$_H$		
`04`$_H$			`0D`$_H$		
`05`$_H$			`0E`$_H$		
`06`$_H$			`0F`$_H$		
`07`$_H$			`10`$_H$		
`08`$_H$			`11`$_H$		
`09`$_H$			`12`$_H$		

2.9 Cell Attributes (BIFF2)

All cell records in BIFF2 contain a cell attribute field with a size of 3 bytes. They contain an index to an XF record (→5.37) and some repeated contents of the referenced XF record. The XF index field has a size of only 6 bits, so the index range is 0-63. If an index >62 is used, the XF index field always contains the vaue 63, and an IXFE record (→5.20) occurs in front of the cell record. It contains the correct index of the XF record.

Cell attributes field (3 bytes), BIFF2:

Offset	Size	Contents		
0	1	Cell protection and XF index:		
		Bit	**Mask**	**Contents**
		5-0	$3F_H$	Index to XF record (→5.37). The value $3F_H$ (63) indicates a preceding IXFE record (→5.20).
		6	40_H	1 = Cell is locked
		7	80_H	1 = Formula is hidden
1	1	Indexes to FORMAT and FONT records:		
		Bit	**Mask**	**Contents**
		5-0	$3F_H$	Index to FORMAT record (→5.16)
		7-6	$C0_H$	Index to FONT record (→5.15)
2	1	Cell style:		
		Bit	**Mask**	**Contents**
		2-0	07_H	XF_HOR_ALIGN – Horizontal alignment (→5.37.1)
		3	08_H	1 = Cell has left black border
		4	10_H	1 = Cell has right black border
		5	20_H	1 = Cell has top black border
		6	40_H	1 = Cell has bottom black border
		7	80_H	1 = Cell has shaded background

3 Formulas

3.1 Common Structure

Formulas are stored as part of a record, for instance inside of a FORMULA record or a NAME record. The common format of a formula is as follows:

Formula in BIFF2:

Offset	Size	Contents
0	1	Size of the following formula data (RPN token array) (sz)
1	sz	Formula data (RPN token array)

Formula in BIFF3-BIFF8:

Offset	Size	Contents
0	2	Size of the following formula data (sz)
2	sz	Formula data (RPN token array)

The contents of a formula are stored in the Reverse-Polish Notation (RPN). This means, first occur all operands of an operation, followed by the respective operator. The operands and operators are called tokens. For instance the simple term 1+2 consists of 3 tokens. Written in RPN the formula is converted to the token list „1“, „2“, „+“. During parsing such an expression operands are pushed onto a stack. An operator pops the needed number of operands from stack, performs the operation and pushes the result back onto the stack.

Other examples for RPN token arrays:

Formula	Token array	Parsing result
2·4+5	2, 4, „·“, 5, „+“	The „·“ pops 4 and 2 and pushes 8, the „+“ pops 5 and 8 and pushes 40. That is the result.
2+4·5	2, 4, 5, „·“, „+“	The „·“ pops 5 and 4 and pushes 20, the „+“ pops 20 and 2 and pushes 22. That is the result.

A token can be a simple integer or floating point value, a string constant, a cell reference or cell range reference or an operator. A token is stored as follows:

Offset	Size	Contents
0	1	Token identifier
[1]	var.	(optional) Additional data for the token

Example of the formula for the term 1+2:

Offset	Size	Data	Name	Comment
0	2	0007$_H$	sz	Size of the following formula data
2	1	1E$_H$	ptgInt	} Integer value token
3	2	0001$_H$		
5	1	1E$_H$	ptgInt	} Integer value token
6	2	0002$_H$		
8	1	03$_H$	ptgAdd	Addition operator

In the following token descriptions, only the additional data following the token identifier is described.

3.2 Operators

There are 3 types of operators:

- Unary operators like the minus sign that negates a value. These operators pop the topmost operand from the stack.
- Binary operators like addition or multiplication. These operators pop the two topmost operands from the stack.
- Function operators represent the sheet functions of Excel. They operate on different numbers of topmost operands on the stack. Either the function expects a fixed number of operands (for instance SIN expects one operand) or a variable number of operands given in the function token (for instance SUM is able to process from 0 to 30 operands).

3.3 Reference Classes

Some of the tokens (especially function operators and operand tokens) exist in 3 different versions: reference class token, value class token and array class token. The token class depends on which type of data the involved operator expects. Sometimes only 1 or 2 token classes make sense.

- Reference class token: The reference itself, independent of the cell contents.
- Value class token: A dereferenced value.
- Array class token: A matrix reference to a cell range.

The structure of the 8-bit operand token identifier is described in the following table.

Bit	Mask	Contents
4-0	$1F_H$	Basic token identifier
6-5	60_H	01_2 = Reference class token (token range 20_H-$3F_H$) 10_2 = Value class token (token range 40_H-$5F_H$) 11_2 = Array class token (token range 60_H-$7F_H$)
7	80_H	0_2 (zero)

The class of an operand token is marked in its name: The names of value class tokens contain a trailing „V“ and the names of array class tokens a trailing „A“.

Examples for the different token classes:

- Reference class token: The formula `=ROW(A1)` returns 1, regardless of the content of A1. Cell reference token is ptgRef (24_H).
- Value class token: The formula `=A1+1` returns the value of the cell A1, increased by 1. Cell reference token is ptgRefV (44_H).
- Array class token: The formula `=MDETERM(A1:C3)` returns the determinant of the values inside of the matrix range A1:C3. Area reference token is ptgAreaA (65_H).

3.4 Encoding of Cell References in Tokens

3.4.1 Cell references in BIFF2-BIFF7

In the BIFF versions up to BIFF5, it is possible to use 16384 rows (2^{14}). A cell reference contains the row index as a 16-bit-value (zero-based, 0-16383), the column index as an 8-bit-value (zero-based, 0-255) and two flags. The flags specify whether the row or column index is absolute or relative.

Contents of the row index (16-bit-value), BIFF2-BIFF7:

Bit	Mask	Contents	
13-0	$3FFF_H$	Index to row (0-16383)	
14	4000_H	0 = Absolute column reference	1 = Relative column reference
15	8000_H	0 = Absolute row reference	1 = Relative row reference

Example: The reference B$6 has the absolute row index 5 and the relative column index 1. The value of the encoded row index is 4005_H (row 6, column is relative). The value of the column index is 01_H (column B).

3.4.2 Cell references in BIFF8

From BIFF8 on 65536 (2^{16}) rows are available. Therefore the column index field expands to a 16-bit-value and contains the relative flags.

Contents of the column index (16-bit-value), BIFF8:

Bit	Mask	Contents	
7-0	$00FF_H$	Index to column (0-255)	
14	4000_H	0 = Absolute column reference	1 = Relative column reference
15	8000_H	0 = Absolute row reference	1 = Relative row reference

Example: The reference B$6 has the absolute row index 5 and the relative column index 1. The value of the encoded column index is 4001_H (column B, column is relative). The value of the row index is 0005_H (row 6).

3.5 Token Overview

Following a list of all tokens, separated into several token classes and ordered by token identifier.

3.5.1 Unary operator tokens

Token ID	Token name	Description
12_H	ptgUplus	Unary plus
13_H	ptgUminus	Unary minus
14_H	ptgPercent	Percent sign

3.5.2 Binary operator tokens

Token ID	Token name	Description
03_H	ptgAdd	Addition
04_H	ptgSub	Subtraction
05_H	ptgMul	Multiplication
06_H	ptgDiv	Division
07_H	ptgPower	Exponentiation
08_H	ptgConcat	Concatenation
09_H	ptgLT	Less than
$0A_H$	ptgLE	Less than or equal
$0B_H$	ptgEQ	Equal
$0C_H$	ptgGE	Greater than or equal
$0D_H$	ptgGT	Greater than
$0E_H$	ptgNE	Not equal
$0F_H$	ptgIsect	Cell range intersection
10_H	ptgUnion	Cell range union
11_H	ptgRange	Cell range

3.5.3 Function operator tokens

Token ID	Token name	Description
21_H 41_H 61_H	ptgFunc	Function with fixed number of arguments
22_H 42_H 62_H	ptgFuncVar	Function with variable number of arguments

3.5.4 Constant operand tokens

Token ID	Token name	Description
16_H	ptgMissArg	Missing argument
17_H	ptgStr	String constant
$1C_H$	ptgErr	Error value
$1D_H$	ptgBool	Boolean value
$1E_H$	ptgInt	Integer value
$1F_H$	ptgNum	Floating-point number

3.5.5 Operand tokens

Token ID	Token name	Description
20$_H$ 40$_H$ 60$_H$	ptgArray	Array constant
23$_H$ 43$_H$ 63$_H$	ptgName	Internal defined name
24$_H$ 44$_H$ 64$_H$	ptgRef	2D cell reference
25$_H$ 45$_H$ 65$_H$	ptgArea	2D area reference
2A$_H$ 4A$_H$ 6A$_H$	ptgRefErr	Deleted 2D cell reference
2B$_H$ 4B$_H$ 6B$_H$	ptgAreaErr	Deleted 2D area reference
39$_H$ 59$_H$ 79$_H$	ptgNameX	External name
3A$_H$ 5A$_H$ 7A$_H$	ptgRef3d	3D cell reference
3B$_H$ 5B$_H$ 7B$_H$	ptgArea3d	3D area reference
3C$_H$ 5C$_H$ 7C$_H$	ptgRefErr3d	Deleted 3D cell reference
3D$_H$ 5D$_H$ 7D$_H$	ptgAreaErr3d	Deleted 3D area reference
2do: more		

3.6 Unary Operator Tokens

Unary operators perform an operation with the topmost operand from stack. The tokens do not contain any additional data.

3.6.1 ptgUplus (12$_H$)

Unary plus operator. This operator has no effect on the operand.

Example: `+1` returns 1.

3.6.2 ptgUminus (13$_H$)

Unary minus operator. Negates the operand.

Example: `-1` returns -1.

3.6.3 ptgPercent (14$_H$)

Percent sign. Divides the operand by 100.

Example: `1%` returns 0.01.

3.7 Binary Operator Tokens

Binary operators perform an operation with the two topmost operands from stack. The tokens do not contain any additional data.

3.7.1 ptgAdd (03$_H$)

Addition operator. Adds the operands.

Example: `3+2` returns 5.

3.7.2 ptgSub (04_H)

Subtraction operator. Subtracts the top operand from the second-to-top operand.
Example: `3-2` returns 1.

3.7.3 ptgMul (05_H)

Multiplication operator. Multiplicates the operands.
Example: `3*2` returns 6.

3.7.4 ptgDiv (06_H)

Division operator. Divides the second-to-top operand by the top operand.
Example: `3/2` returns 1.5.

3.7.5 ptgPower (07_H)

Exponentiation operator. Raises the second-to-top operand to the power of the top operand.
Example: `3^2` returns 9.

3.7.6 ptgConcat (08_H)

Concatenation operator. Appends the top operand to the second-to-top operand.
Example: `"ABC"&"DEF"` returns "ABCDEF“.

3.7.7 ptgLT (09_H)

Less than operator. Returns TRUE if the second-to-top operand is less than the top operand.
Example: `3<2` returns FALSE.

3.7.8 ptgLE ($0A_H$)

Less than or equal operator. Returns TRUE if the second-to-top operand is less than or equal to the top operand.
Example: `3<=2` returns FALSE.

3.7.9 ptgEQ ($0B_H$)

Equality operator. Returns TRUE if the operands are equal.
Example: `3=2` returns FALSE.

3.7.10 ptgGE ($0C_H$)

Greater than or equal operator. Returns TRUE if the second-to-top operand is greater than or equal to the top operand.
Example: `3>=2` returns TRUE.

3.7.11 ptgGT ($0D_H$)

Greater than operator. Returns TRUE if the second-to-top operand is greater than the top operand.
Example: `3>2` returns TRUE.

3.7.12 ptgNE (0E$_H$)

Inequality operator. Returns TRUE if the operands are not equal.

Example: `3<>2` returns TRUE.

3.7.13 ptgIsect (0F$_H$)

Intersection operator, represented by the space sign. Returns the intersected range of two ranges.

Example: `A1:B3 B2:C3` returns B2:B3.

3.7.14 ptgUnion (10$_H$)

Union operator, represented by the comma sign (for instance english Excel) or semicolon (for instance german Excel). Returns the union of two ranges.

Example: `(A1:A2,A2:A3)` will be handled as one parameter (useful for function parameters).

3.7.15 ptgRange (11$_H$)

Range operator, represented by the colon sign. Returns the rectangular range formed by two ranges. This token occurs for instance by using defined names.

Example: `namedcell:D5`.

3.8 Function Operator Tokens

3.8.1 ptgFunc (21$_H$), ptgFuncV (41$_H$), ptgFuncA (61$_H$)

This token contains the index to a function with fixed number of arguments.

Token ptgFunc, BIFF2-BIFF3:

Offset	Size	Contents
0	1	Index to a sheet function

Token ptgFunc, BIFF4-BIFF8:

Offset	Size	Contents
0	2	Index to a sheet function

3.8.2 ptgFuncVar (21$_H$), ptgFuncVarV (41$_H$), ptgFuncVarA (61$_H$)

This token contains the index to a function with variable number of arguments.

Token ptgFuncVar, BIFF2-BIFF3:

Offset	Size	Contents
0	1	Number of arguments
1	1	Index to a sheet function

Token ptgFuncVar, BIFF4-BIFF8:

Offset	Size	Contents
0	1	Number of arguments
1	2	Index to a sheet function

3.9 Constant Operand Tokens

3.9.1 ptgMissArg (16_H)

A missing argument in a function argument list is stored as a ptgMissArg token. This token does not contain any additional data.

Example: `SUM(1,,3)` – second argument is missing and represented by a ptgMissArg token.

3.9.2 ptgStr (17_H)

This token contains a string constant. The maximum length of the string is 253 characters in BIFF2 (due to the limitation of 255 bytes per formula) and 255 characters in BIFF3-BIFF7.

Token ptgStr, BIFF2-BIFF7:

Offset	Size	Contents
0	var.	Byte string, 8-bit string length (➜2.1)

Token ptgStr, BIFF8:

Offset	Size	Contents
0	var.	Unicode string, 16-bit string length, option flags occur always (➜2.2)

Example: `"ABC"`.

3.9.3 ptgErr ($1C_H$)

This token contains an error code.

Offset	Size	Contents
0	1	Error code (➜2.4)

3.9.4 ptgBool ($1D_H$)

This token contains a boolean value (TRUE or FALSE).

Offset	Size	Contents
0	1	0 = FALSE, 1 = TRUE

3.9.5 ptgInt ($1E_H$)

This token contains an unsigned 16-bit-integer value in the range from 0 to 65535.

Offset	Size	Contents
0	2	Unsigned integer value

3.9.6 ptgNumber ($1F_H$)

This token contains an IEEE floating-point number.

Offset	Size	Contents
0	8	IEEE floating-point number

3.10 Operand Tokens

3.10.1 ptgArray (20_H), ptgArrayV (40_H), ptgArrayA (60_H)

This token contains an array constant. For instance the 2x1 matrix `{1;2}` is an array constant. The values of the array constant do not follow the token identifier but are stored behind the complete token array.

Token ptgArray, BIFF2-BIFF8:

Offset	Size	Contents
0	7	Not used

The constants of the array are stored row by row behind the formula in a list. The length of this list has not been added to the leading formula size field.

Array constant list, BIFF2-BIFF7:

Offset	Size	Contents
0	1	Number of columns (nc). The value 0 means 256 columns.
1	2	Number of rows (nr)
3	var.	List of nc·nr cached values (→2.5)

Array constant list, BIFF8:

Offset	Size	Contents
0	1	Number of columns decreased by 1 (nc)
1	2	Number of rows decreased by 1 (nr)
3	var.	List of (nc+1)·(nr+1) cached values (→2.5)

3.10.2 ptgName (23_H), ptgNameV (43_H), ptgNameA (63_H)

This token contains the one-based index to a NAME record (→5.25). In BIFF2-BIFF4 this could be the index to an EXTERNNAME record (→5.12) too. From BIFF5 on an external name is represented by the token ptgNameX (→3.10.7).

Token ptgName, BIFF2:

Offset	Size	Contents
0	2	One-based index to NAME record (→5.25) or EXTERNNAME record (→5.12)
2	5	Not used

Token ptgName, BIFF3-BIFF4:

Offset	Size	Contents
0	2	One-based index to NAME record (→5.25) or EXTERNNAME record (→5.12)
2	8	Not used

Token ptgName, BIFF5/BIFF7:

Offset	Size	Contents
0	2	One-based index to NAME record (→5.25)
2	12	Not used

Token ptgName, BIFF8:

Offset	Size	Contents
0	2	One-based index to NAME record (→5.25)
2	2	Not used

3.10.3 ptgRef (24$_H$), ptgRefV (44$_H$), ptgRefA (64$_H$)

This token contains the reference to a cell in the same sheet.
Token ptgRef, BIFF2-BIFF7:

Offset	Size	Contents
0	2	Index to row and relative flags (→3.4.1)
2	1	Index to column

Token ptgRef, BIFF8:

Offset	Size	Contents
0	2	Index to row
2	2	Index to column and relative flags (→3.4.2)

3.10.4 ptgArea (25$_H$), ptgAreaV (45$_H$), ptgAreaA (65$_H$)

This token contains the reference to a rectangular cell range in the same sheet.
Token ptgArea, BIFF2-BIFF7:

Offset	Size	Contents
0	2	Index to first row and relative flags (→3.4.1)
2	2	Index to last row and relative flags (→3.4.1)
4	1	Index to first column
5	1	Index to last column

Token ptgArea, BIFF8:

Offset	Size	Contents
0	2	Index to first row
2	2	Index to last row
4	2	Index to first column and relative flags (→3.4.2)
6	2	Index to last column and relative flags (→3.4.2)

3.10.5 ptgRefErr (2A$_H$), ptgRefErrV (4A$_H$), ptgRefErrA (6A$_H$)

This token contains the last reference to a deleted cell in the same sheet. The structure is equal to the token ptgRef (→3.10.3).

3.10.6 ptgAreaErr (2B$_H$), ptgAreaErrV (4B$_H$), ptgAreaErrA (6B$_H$)

This token contains the last reference to a deleted rectangular cell range in the same sheet. The structure is equal to the token ptgArea (→3.10.4).

3.10.7 ptgNameX (39$_H$), ptgNameXV (59$_H$), ptgNameXA (79$_H$) (BIFF5-BIFF8)

This token contains the index to a NAME or EXTERNNAME record. It occurs by using internal or external names, AddIn functions, DDE links or linked OLE objects. See →4.5.2 for details about references in BIFF5/BIFF7 and →4.5.3 for BIFF8.

Token ptgNameX, BIFF5/BIFF7:

Offset	Size	Contents
0	2	One-based index to EXTERNSHEET record (→5.13). A negative value indicates the own workbook. In this case a NAME record is indexed below. The absolute value indexes to the EXTERNSHEET record that contains the sheet name.
2	8	Not used
10	2	One-based index to NAME record (→5.25) or EXTERNNAME record (→5.12)
12	12	Not used

Token ptgNameX, BIFF8:

Offset	Size	Contents
0	2	Index to REF entry in EXTERNSHEET record (→5.13)
2	2	One-based index to NAME record (→5.25) or EXTERNNAME record (→5.12)
4	2	Not used

3.10.8 ptgRef3d (3A$_H$), ptgRef3dV (5A$_H$), ptgRef3dA (7A$_H$) (BIFF5-BIFF8)

This token contains a 3D reference or an external reference to a cell. See →4.5.2 for details about references in BIFF5/BIFF7 and →4.5.3 for BIFF8.

Token ptgRef3d, BIFF5/BIFF7:

Offset	Size	Contents
0	2	One-based index to EXTERNSHEET record (→5.13). A negative value indicates a 3D reference to the own workbook. The absolute value indexes to the EXTERNSHEET record that contains the first sheet name.
2	8	Not used
10	2	3D reference: Index of first referenced sheet; External reference: Not used
12	2	3D reference: Index of last referenced sheet; External reference: Not used
14	2	Index to row and relative flags (→3.4.1)
16	1	Index to column

Token ptgRef3d, BIFF8:

Offset	Size	Contents
0	2	Index to REF entry in EXTERNSHEET record (→5.13)
2	2	Index to row
4	2	Index to column and relative flags (→3.4.2)

3.10.9 ptgArea3d (3B$_H$), ptgArea3dV (5B$_H$), ptgArea3dA (7B$_H$) (BIFF5-BIFF8)

This token contains a 3D reference or an external reference to a rectangular cell range. See →4.5.2 for details about references in BIFF5/BIFF7 and →4.5.3 for BIFF8.

Token ptgArea3d, BIFF5/BIFF7:

Offset	Size	Contents
0	2	One-based index to EXTERNSHEET record (→5.13). A negative value indicates a 3D reference to the own workbook. The absolute value indexes to the EXTERNSHEET record that contains the first sheet name.
2	8	Not used
10	2	3D reference: Index of first referenced sheet; External reference: Not used
12	2	3D reference: Index of last referenced sheet; External reference: Not used
14	2	Index to first row and relative flags (→3.4.1)
16	2	Index to last row and relative flags (→3.4.1)
18	1	Index to first column
19	1	Index to last column

Token ptgArea3d, BIFF8:

Offset	Size	Contents
0	2	Index to REF entry in EXTERNSHEET record (→5.13)
2	2	Index to first row
4	2	Index to last row
6	2	Index to first column and relative flags (→3.4.2)
8	2	Index to last column and relative flags (→3.4.2)

3.10.10 ptgRefErr3d (3C$_H$), ptgRefErr3dV (5C$_H$), ptgRefErr3dA (7C$_H$) (BIFF5-BIFF8)

This token contains the last 3D reference or external reference to a deleted cell. The structure is equal to the token ptgRef3d (→3.10.8).

3.10.11 ptgAreaErr3d (3D$_H$), ptgAreaErr3dV (5D$_H$), ptgAreaErr3dA (7D$_H$) (BIFF5-BIFF8)

This token contains the last 3D reference or external reference to a deleted rectangular cell range. The structure is equal to the token ptgArea3d (→3.10.9).

4 Worksheet/Workbook Structure

In an Excel file, some complex features are splitted into several records. To keep these features consistent, the position and order of the records is very important. This chapter contains details about the correct combination of the records inside of the stream. The internal structure of the records is described in chapter 5.

4.1 Worksheet Stream (BIFF2-BIFF4)

The whole worksheet file consists of the worksheet stream. All records of the worksheet are enclosed by a leading BOF record (➜5.4) and a trailing EOF record (➜5.10). The sheet contents section contains all information about the worksheet, for instance sheet dimension, view settings, a font list, a list of defined names and external references, of course the contents and formats of all cells, row heights, column widths, drawing objects, chart objects, etc.

Common structure of a worksheet stream:

BOF	Type = worksheet
	Sheet contents
EOF	

4.2 Workbook Stream (BIFF4)

The whole BIFF4 workbook file consists of the workbook stream. It contains a workbook globals section and a list of the worksheets. The sheets are embedded into the outer pair of BOF/EOF records. The workbook globals section contains common information about the workbook, for instance text encoding, global view settings or a list of all sheet names. Additionally, in each workbook a SHEETSOFFSET record (➜5.32) is present. The data of the sheets is stored in worksheet substreams. Each substream is preceded by a SHEETHDR record (➜5.31) which contains the name of the sheet and the size of the following substream. The SHEETSOFFSET record mentioned above contains the stream position of the first SHEETHDR record. The substreams have the same structure as described in chapter 4.1. Note: In this context the term „substream“ is only a sequence of records and not a storage sub stream of OLE2 storages.

Common structure of a workbook stream with two sheets, BIFF4:

BOF	Type = workbook globals
	Workbook globals
SHEETSOFFSET	Position of the first SHEETHDR record
	Workbook globals
SHEETHDR	Sheet name = „Sheet1“, Byte length of following BOF/EOF record block
BOF	Type = worksheet
	Sheet contents
EOF	
SHEETHDR	Sheet name = „Sheet2“, Byte length of following BOF/EOF record block
BOF	Type = worksheet
	Sheet contents
EOF	
EOF	

4.3 Workbook Stream (BIFF5-BIFF8)

From BIFF5 on an Excel document is stored as an OLE2 storage. The workbook stream is located in the root directory of the storage. In BIFF5/BIFF7 it is named „Book“, in BIFF8 „Workbook“. The names are case-sensitive. In difference to the BIFF4 workbook stream, the worksheet substreams are appended to the workbook globals section, not embedded. The workbook global section and sheet contents section have similar contents as described for BIFF4 workbooks (→4.2).

Common structure of a workbook stream with two sheets, BIFF5-BIFF8:

BOF	Type = workbook globals
	Workbook globals
EOF	
BOF	Type = worksheet
	Sheet contents
EOF	
BOF	Type = worksheet
	Sheet contents
EOF	

4.4 Shared String Table (BIFF8)

A BIFF8 workbook collects all strings of all text cells in a global list, the shared string table (SST). This table is located in the workbook globals section in the record SST (→5.33). An SST record is followed by an EXTSST record (→5.14) which stores stream positions for a string hash table. Text cells are represented by LABELSST records (→5.22) which contain indexes to the shared string table. For reading Excel files only the SST record and the LABELSST records are important.

Example: A workbook contains anywhere the strings „AAA“, „BBB“ and „CCC“.

BOF	Type = workbook globals
	Workbook globals
SST	String 0 = „AAA“ String 1 = „BBB“ String 2 = „CCC“
EXTSST	
	Workbook globals
EOF	
BOF	Type = worksheet
	Sheet contents
LABELSST	String = 0
LABELSST	String = 2
	Sheet contents
LABELSST	String = 1
LABELSST	String = 0
	Sheet contents
EOF	

4.5 Internal and External References

This chapter describes all types of 3D and external references. In detail, this could be:

- a reference to a cell or a cell range of another sheet in the same workbook (3D reference),
- a reference to a cell or a cell range of a sheet in another workbook (external reference),
- a reference to a global or local defined name (internal name),
- a reference to a defined name in another workbook (external name),
- an AddIn function,
- a DDE link,
- an OLE object link.

For external references and external names a combination of XCT and CRN records occurs which store values of cells of the document. In the case the document cannot be found these values will be used to get the result of an external reference. An XCT record (→5.36) contains the number of following CRN records. A CRN record (→5.7) stores the contents of one cell or a sequence of cells of one row. Fragmentary cell ranges or cell ranges spanning over more than one row are splitted into several CRN records. 3D references do not use these records because the referenced cells are located in the own document.

4.5.1 References in BIFF2-BIFF4

2do

4.5.2 References in BIFF5/BIFF7

The document names and sheet names of references are stored in a list of EXTERNSHEET records. Each worksheet contains an EXTERNSHEET list with documents referenced from this sheet. Formulas in the sheet use indexes to the EXTERNSHEET list.

The XCT and CRN records occur behind the last EXTERNNAME record as far as they exist, otherwise directly behind the respective EXTERNSHEET record.

• External and 3D references

External and 3D references are represented in a formula by the tokens ptgRef3d (→3.10.8) or ptgArea3d (→3.10.9). These tokens contain an index to an EXTERNSHEET record located in the own worksheet and indexes to the first and last referenced sheet.

For 3D references, the tokens contain a negative EXTERNSHEET index, indicating a reference into the own workbook. The absolute value is the <u>one-based</u> index of the EXTERNSHEET record that contains the name of the first sheet. If the referenced sheets do not exist anymore, these tokens contain the sheet indexes `FFFF`$_H$ (deleted 3D reference).

Each external reference contains the <u>one-based</u> index to an EXTERNSHEET record. The sheet indexes of the tokens are not used.

Example: A document with 7 sheets (named from „Sheet1“ to „Sheet7“) contains the formulas
`=Sheet2!A1`,
`=SUM(Sheet4:Sheet6!A1:B3)`,
`=SUM([example.xls]ExtSheet1!A1:B2)` (contents: A1=1.11, B1=2.22, A2=3.33, B2=4.44),
`=[example.xls]ExtSheet3!A1` (contents: „ABCD“) and
`=Sheet8!A1`.

EXTERNSHEET 1	Name = „Sheet2“
EXTERNSHEET 2	Name = „Sheet4“
EXTERNSHEET 3	Name = „Sheet6“
EXTERNSHEET 4	Name = „[example.xls]ExtSheet1“
XCT	Number of CRN = 2
CRN 0	Cell range = A1:B1, contents = 1.11, 2.22
CRN 1	Cell range = A2:B2, contents = 3.33, 4.44
EXTERNSHEET 5	Name = „[example.xls]ExtSheet3“
XCT	Number of CRN = 1
CRN 0	Cell range = A1, contents = „ABCD“
EXTERNSHEET 6	Name = „Sheet8“

• Internal names

2do

• External names

2do

• AddIn functions

2do

• DDE links, OLE object links

2do

4.5.3 References in BIFF8

The main data of all types of references is stored in a list inside of the workbook globals section. All formulas use only indexes to use specific references. In BIFF8 each referenced document is represented by a SUPBOOK record (➜5.35). A SUPBOOK contains the name of the document and the names of the sheets of the document. After the last SUPBOOK occurs only one EXTERNSHEET record (➜5.13). It contains a list with indexes to the SUPBOOKs for each used reference anywhere in the document. Formulas use indexes into this EXTERNSHEET list.

For the following examples an external document „example.xls“ is used. It contains 3 sheets named „ExtSheet1“, „ExtSheet2“ and „ExtSheet3“.

Example: A document contains (among other references) the two formulas
`=[example.xls]ExtSheet2!A1` and
`=[example.xls]ExtSheet1!A1`.

	Workbook globals
SUPBOOK 0	Any content
SUPBOOK 1	Document = „example.xls“ Sheet 0 = „ExtSheet1“ Sheet 1 = „ExtSheet2“ Sheet 2 = „ExtSheet3“
SUPBOOK 2	Any content
EXTERNSHEET	REF 0 = any reference REF 1 = {SUPBOOK = 1, sheet range = 1…1} REF 2 = any reference REF 3 = {SUPBOOK = 1, sheet range = 0…0} REF 4 = any reference
	Workbook globals

The first formula uses REF 1 in the EXTERNSHEET record. REF 1 refers to SUPBOOK 1 and sheet range 1…1. This means, the document „example.xls“ is used (document of SUPBOOK 1) and the name of the sheet is „ExtSheet2“ (sheet 1 of SUPBOOK 1). In the same way, the second formula uses REF 3 in the EXTERNSHEET record. All list entries inside of the EXTERNSHEET record are unique. For instance all formulas in the workbook referring to sheet „ExtSheet2“ of the document „example.xls“ use REF 1. All other SUPBOOKs and REFs are placeholders for other references in this example.

The XCT and CRN records occur behind the EXTERNNAME records as far as they exist, otherwise directly behind the respective SUPBOOK record.

• External and 3D references

The SUPBOOK for the own document has a special format: It contains only the number of all sheets and the value 0401_H instead of the sheet names. The sheet range indexes in the EXTERNSHEET record refer to the position of the sheets (zero-based). If a referenced sheet does not exist anymore, the sheet index $FFFF_H$ will occur (deleted 3D reference).

Example: A document with 7 sheets (named from „Sheet1“ to „Sheet7“) contains the formulas
`=Sheet2!A1`,
`=SUM(Sheet4:Sheet6!A1:B3)`,
`=SUM([example.xls]ExtSheet1!A1:B2)` (contents: A1=1.11, B1=2.22, A2=3.33, B2=4.44),
`=[example.xls]ExtSheet3!A1` (contents: „ABCD“) and
`=Sheet8!A1`.

SUPBOOK 0	Number of sheets: 7 `0401`$_H$ (own workbook)
SUPBOOK 1	Document = „example.xls“ Sheet 0 = „ExtSheet1“ Sheet 1 = „ExtSheet2“ Sheet 2 = „ExtSheet3“
XCT	Number of CRN = 2, sheet = 0 (ExtSheet1)
CRN 0	Cell range = A1:B1, contents = 1.11, 2.22
CRN 1	Cell range = A2:B2, contents = 3.33, 4.44
XCT	Number of CRN = 1, sheet = 2 (ExtSheet3)
CRN 0	Cell range = A1, contents = „ABCD“

EXTERNSHEET	REF 0 = {SUPBOOK = 0, sheet range = 1…1}
	REF 1 = {SUPBOOK = 0, sheet range = 3…5}
	REF 2 = {SUPBOOK = 1, sheet range = 0…0}
	REF 3 = {SUPBOOK = 1, sheet range = 1…1}
	REF 4 = {SUPBOOK = 0, sheet range = FFFF$_H$…FFFF$_H$}

Inside of the first formula the cell reference is represented by the token ptgRef3d (→3.10.8). The second formula contains the token ptgArea3d (→3.10.9).

• Internal names

All internal names are stored in a list of NAME records (→5.25) that follows the EXTERNSHEET record. There exist two types of internal names: global names which are valid in the whole workbook and local names which are attached to a specific sheet. For instance the local name „MyCell“ of the sheet „Sheet1“ can be used from everywhere in the workbook by entering `=Sheet1!MyCell`. Each NAME record contains the name itself and a one-based sheet index. The index zero indicates a global name. If a SUPBOOK contains local names, a special REF entry will be created in the EXTERNSHEET record. It contains the index to the SUPBOOK and the sheet range FFFE$_H$…FFFE$_H$.

Example for internal names: A document contains
- The global name „GlobalName“,
- The local names „Sheet1!Name“ and „Sheet2!Name“ and
- In Sheet1 the formulas `=GlobalName`, `=Name`, `=Sheet1!Name` and `=Sheet2!Name`.

SUPBOOK 0	Number of sheets: 3 `0401`$_H$ (own workbook)

EXTERNSHEET	REF 0 = {SUPBOOK = 0, sheet range = 0…0}
	REF 1 = {SUPBOOK = 0, sheet range = FFFE$_H$…FFFE$_H$}

NAME 1	Name = „GlobalName“, sheet = 0 (Global)
NAME 2	Name = „Name“, sheet = 1 (Sheet1)
NAME 3	Name = „Name“, sheet = 2 (Sheet2)

Inside of the formula a global name or a local name of the own sheet is represented by the token ptgName (→3.10.2) with an one-based index to the NAME record list. The first formula in the example above contains the token ptgNameV with index 1 and the second formula the same token with index 2.

Local names from other sheets are represented by the token ptgNameX (→3.10.7) with an index to the special REF entry of the EXTERNSHEET record and an index to the NAME record list. The third formula contains the token ptgNameX with the value {REF = 1, Name = 2} and the last formula the same token with the value {REF = 1, Name = 3}. Ref 1 refers to SUPBOOK 0 and Name 2 or Name 3 refer to the respective NAME records.

• External names

In Excel formulas can use names located in another workbook. In this case for each name an EXTERNNAME record (➜5.12) after the SUPBOOK record occurs. The EXTERNNAME record contains the name itself and the one-based index to the sheet. Again the index zero indicates a global name. If a SUPBOOK contains external names, a special REF entry will be created in the EXTERNSHEET record. It contains the index to the SUPBOOK and the sheet range $\text{FFFE}_H \ldots \text{FFFE}_H$.

Example: A document contains the formulas
`=example.xls!GlobalName` (location: ExtSheet1!B22; contents: 22),
`=[example.xls]ExtSheet3!Name` (location: ExtSheet3!C33; contents: „ABCD“) and
`=[example.xls]ExtSheet1!Name` (location: ExtSheet1!A11; contents: 11).

SUPBOOK 0	Document = „example.xls“ Sheet 0 = „ExtSheet1“ Sheet 1 = „ExtSheet2“ Sheet 2 = „ExtSheet3“
EXTERNNAME 1	Name = „GlobalName“, sheet = 0 (Global)
EXTERNNAME 2	Name = „Name“, sheet = 3 (ExtSheet3)
EXTERNNAME 3	Name = „Name“, sheet = 1 (ExtSheet1)
XCT	Number of CRN = 2, sheet = 0 (ExtSheet1)
CRN 0	Cell range = A11, contents = 11
CRN 1	Cell range = B22, contents = 22
XCT	Number of CRN = 1, sheet = 2 (ExtSheet3)
CRN 0	Cell range = C33, contents = „ABCD“
EXTERNSHEET	REF 1 = {SUPBOOK = 0, sheet range = $\text{FFFE}_H \ldots \text{FFFE}_H$}

Inside of a formula an external name is represented by the token ptgNameX (➜3.10.7). It contains the index to the special REF entry inside of the EXTERNSHEET record and the index to a EXTERNNAME record (one-based). The second formula in the example above contains the token ptgNameXV with the value {REF = 0, ExtName = 2}. REF 1 refers to SUPBOOK 0 and ExtName 2 refers to EXTERNNAME 2 (of SUPBOOK 0).

• AddIn functions

AddIn functions are stored similar to external names. If a formula uses an AddIn function, a special SUPBOOK containing only the value $\texttt{3A01}_H$ will occur. Behind of this SUPBOOK the names of all used AddIn functions are listed, each inside of an EXTERNNAME record. A special REF entry with the sheet range $\text{FFFE}_H \ldots \text{FFFE}_H$ will be inserted into the EXTERNSHEET reference list.

Example: A document contains the formulas `=ISODD(1)` and `=ISEVEN(1)`.

SUPBOOK 0	$\texttt{3A01}_H$ (AddIn)
EXTERNNAME 1	Name = „ISODD“
EXTERNNAME 2	Name = „ISEVEN“
EXTERNSHEET	REF 0 = {SUPBOOK = 0, sheet range = $\text{FFFE}_H \ldots \text{FFFE}_H$}

• DDE links, OLE object links

DDE links and OLE object links expect the name of the server application (DDE) or the class name (OLE) and the name of a source document. These items are encoded in a SUPBOOK record. The SUPBOOK is followed by EXTERNNAME records with additional data of the links. An EXTERNNAME record for a DDE links contains the item (data source range) and an EXTERNNAME record for an OLE object link contains the identifier of the object data storage.

Example: A document contains a DDE link to the range „Sheet1.A1:B2“ inside of the Calc document „example.sxc“ and an OLE object link to the bitmap file „example.bmp“.

SUPBOOK 0	Server application = „soffice“ Document = „example.sxc“

EXTERNNAME 1	Type = DDE link
	Item = „Sheet1.A1:B2“

SUPBOOK 1	Class name = „Package“ Document = „example.bmp“

EXTERNNAME 1	Type = OLE object link
	Storage = 00012345_H (storage name = „LNK00012345“)

EXTERNSHEET	REF 0 = {SUPBOOK = 0, sheet range = $FFFE_H$...$FFFE_H$}
	REF 1 = {SUPBOOK = 1, sheet range = $FFFE_H$...$FFFE_H$}

Inside of a formula a DDE link is represented by the token ptgNameX (→3.10.7). An OLE object link contains a ptgNameX token inside of its OBJ record.

4.6 Array Formulas, Shared Formulas

2do

4.7 Multiple Operations (Table Operations)

2do

4.8 AutoFilter

2do

4.9 Scenarios

2do

4.10 Web Queries (BIFF8)

2do

5 Worksheet/Workbook Records

5.1 Overview, Ordered by Record IDs

Record ID	Record name	Occurs in BIFF versions					
		2	3	4	5	7	8
0000$_H$	DIMENSIONS	x					
0001$_H$	BLANK	x					
0002$_H$	INTEGER	x					
0003$_H$	NUMBER	x					
0004$_H$	LABEL	x					
0005$_H$	BOOLERR	x					
0006$_H$	FORMULA	x			x	x	x
0007$_H$	STRING	x					
0009$_H$	BOF	x					
000A$_H$	EOF	x	x	x	x	x	x
0013$_H$	PASSWORD	x	x	x	x	x	x
0016$_H$	EXTERNCOUNT	x	x	x	x	x	
0017$_H$	EXTERNSHEET	x	x	x	x	x	x
0018$_H$	NAME	x			x	x	x
001E$_H$	FORMAT	x	x				
0023$_H$	EXTERNNAME	x			x	x	x
0031$_H$	FONT	x			x	x	x
003C$_H$	CONTINUE	x	x	x	x	x	x
0043$_H$	XF	x					
0044$_H$	IXFE	x					
0051$_H$	DCONREF	x	x	x	x	x	x
0059$_H$	XCT		x	x	x	x	x
005A$_H$	CRN		x	x	x	x	x
008E$_H$	SHEETSOFFSET			x			
008F$_H$	SHEETHDR			x			
0092$_H$	PALETTE	x	x	x	x	x	x
00BD$_H$	MULRK				x	x	x
00BE$_H$	MULBLANK				x	x	x
00E0$_H$	XF				x	x	x
00FC$_H$	SST						x
00FD$_H$	LABELSST						x
00FF$_H$	EXTSST						x
01AE$_H$	SUPBOOK						x
01B8$_H$	HLINK						x
0200$_H$	DIMENSIONS		x	x	x	x	x
0201$_H$	BLANK		x	x	x	x	x
0203$_H$	NUMBER		x	x	x	x	x
0204$_H$	LABEL		x	x	x	x	

Record ID	Record name	Occurs in BIFF versions					
		2	3	4	5	7	8
0205_H	BOOLERR		x	x	x	x	x
0206_H	FORMULA		x				
0207_H	STRING		x	x	x	x	x
0209_H	BOF		x				
0218_H	NAME		x	x			
0223_H	EXTERNNAME		x	x			
0231_H	FONT		x	x			
0243_H	XF		x				
$027E_H$	RK		x	x	x	x	x
0406_H	FORMULA			x			
0409_H	BOF			x			
$041E_H$	FORMAT			x	x	x	x
0443_H	XF			x			
0800_H	SCREENTIP						x
0809_H	BOF				x	x	x
2do: more							

5.2 Overview, Ordered by Record Names

Record ID	Record name	Occurs in BIFF versions					
		2	3	4	5	7	8
0001$_H$ 0201$_H$	BLANK	x	x	x	x	x	x
0*09$_H$	BOF	x	x	x	x	x	x
0005$_H$ 0205$_H$	BOOLERR	x	x	x	x	x	x
003C$_H$	CONTINUE	x	x	x	x	x	x
005A$_H$	CRN		x	x	x	x	x
0051$_H$	DCONREF	x	x	x	x	x	x
0000$_H$ 0200$_H$	DIMENSIONS	x	x	x	x	x	x
000A$_H$	EOF	x	x	x	x	x	x
0016$_H$	EXTERNCOUNT	x	x	x	x	x	
0023$_H$ 0223$_H$	EXTERNNAME	x	x	x	x	x	x
0017$_H$	EXTERNSHEET	x	x	x	x	x	x
00FF$_H$	EXTSST						x
0031$_H$ 0231$_H$	FONT	x	x	x	x	x	x
001E$_H$ 041E$_H$	FORMAT	x	x	x	x	x	x
0*06$_H$	FORMULA	x	x	x	x	x	x
01B8$_H$	HLINK						x
0002$_H$	INTEGER	x					
0044$_H$	IXFE	x					
0004$_H$ 0204$_H$	LABEL	x	x	x	x	x	
00FD$_H$	LABELSST						x
00BE$_H$	MULBLANK				x	x	x
00BD$_H$	MULRK				x	x	x
0018$_H$ 0218$_H$	NAME	x	x	x	x	x	x
0003$_H$ 0203$_H$	NUMBER	x	x	x	x	x	x
0092$_H$	PALETTE	x	x	x	x	x	x
0013$_H$	PASSWORD	x	x	x	x	x	x
027E$_H$	RK		x	x	x	x	x
0800$_H$	SCREENTIP						x
008F$_H$	SHEETHDR			x			
008E$_H$	SHEETSOFFSET			x			
00FC$_H$	SST						x
0007$_H$ 0207$_H$	STRING	x	x	x	x	x	x
01AE$_H$	SUPBOOK						x
0059$_H$	XCT		x	x	x	x	x
0*43$_H$ 00E0$_H$	XF	x	x	x	x	x	x
2do: more							

5.3 BLANK

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
`0001`H	`0201`H	`0201`H	`0201`H	`0201`H	`0201`H

This record represents an empty cell. It contains the cell address and formatting information.
Record BLANK, BIFF2:

Offset	Size	Contents
0	2	Index to row
2	2	Index to column
4	3	Cell attributes (➜2.9)

Record BLANK, BIFF3-BIFF8:

Offset	Size	Contents
0	2	Index to row
2	2	Index to column
4	2	Index to XF record (➜5.37)

5.4 BOF – Begin of File

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
`0009`H	`0209`H	`0409`H	`0809`H	`0809`H	`0809`H

The BOF record is the first record of a worksheet, the workbook globals section, a chart or a macro sheet.
Record BOF, BIFF2:

Offset	Size	Contents	
0	2	Version	
2	2	Type of the following data:	`0010`H = Worksheet `0020`H = Chart `0040`H = Macro sheet

Record BOF, BIFF3:

Offset	Size	Contents	
0	2	Version	
2	2	Type of the following data:	`0010`H = Worksheet `0020`H = Chart `0040`H = Macro sheet
4	2	Not used	

Record BOF, BIFF4:

Offset	Size	Contents	
0	2	Version	
2	2	Type of the following data:	`0010`H = Worksheet `0020`H = Chart `0040`H = Macro sheet `0100`H = Workbook globals
4	2	Not used	

Record BOF, BIFF5/BIFF7:

Offset	Size	Contents	
0	2	Version	
2	2	Type of the following data:	0005_H = Workbook globals 0006_H = Visual Basic module 0010_H = Worksheet 0020_H = Chart 0040_H = BIFF4 Macro sheet 0100_H = BIFF4 Workbook globals
4	2	Build identifier	
6	2	Build year	

Record BOF, BIFF8:

Offset	Size	Contents	
0	2	Version, contains 0600_H for BIFF8	
2	2	Type of the following data:	0005_H = Workbook globals 0006_H = Visual Basic module 0010_H = Worksheet 0020_H = Chart 0040_H = BIFF4 Macro sheet 0100_H = BIFF4 Workbook globals
4	2	Build identifier	
6	2	Build year	
8	4	File history flags	
12	4	Lowest Excel version that can read all records in this file	

5.5 BOOLERR

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
0005_H	0205_H	0205_H	0205_H	0205_H	0205_H

This record represents a boolean or error value cell.
Record BOOLERR, BIFF2:

Offset	Size	Contents
0	2	Index to row
2	2	Index to column
4	3	Cell attributes (→2.9)
7	1	Boolean or error value, depending on the following byte
8	1	0 = Boolean value; 1 = Error code

Record BOOLERR, BIFF3-BIFF8:

Offset	Size	Contents
0	2	Index to row
2	2	Index to column
4	2	Index to XF record (→5.37)
6	1	Boolean or error value, depending on the following byte
7	1	0 = Boolean value; 1 = Error code

If the value field is a boolean value, it will contain 0 for FALSE and 1 for TRUE. See →2.4 for a list of error codes.

5.6 CONTINUE

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
`003C`$_H$	`003C`$_H$	`003C`$_H$	`003C`$_H$	`003C`$_H$	`003C`$_H$

Everytime the content of a record exceeds the given limits (see table), the record must be splitted. Several CONTINUE records containing the additional data are added after the parent record.

BIFF version	Maximum data size of a record
BIFF2-BIFF7	2080 bytes (2084 bytes including record header)
BIFF8	8224 bytes (8228 bytes including record header)

Record CONTINUE, BIFF2-BIFF8:

Offset	Size	Contents
0	var.	data continuation of the previous record

Unicode strings are splitted in a special way. At the beginning of each CONTINUE record the option flags byte is repeated. Only the character size flag will be set in this flags byte, the Rich-Text flag and the Far-East flag are set to zero.

Attention: In each CONTINUE record it is possible that the character size changes from 8-bit-characters to 16-bit-characters and vice versa. Never an Unicode string is splitted between character count field and option flags field or between option flags field and first character.

Example: The remaining size of a record may be 10 bytes (it has 8214 bytes of data). Now the string „ABCDEFGHØI“ has to be stored in this record. „Ø“ may be a special character with the character code `1234`$_H$. Note: The records are shown with their headers to make the example clearer.

Offset	Size	Contents	Description
0	2		Any record identifier
2	2	`2020`$_H$ (8224)	Record data size
4	8214		Any data
8218	2	`000A`$_H$ (10)	Unicode string character count
8220	1	`00`$_H$	Unicode string option flags (8-bit-characters)
8221	7	`41`$_H$ `42`$_H$ … `47`$_H$	8-bit-character array „ABCDEFG“
8228	2	`003C`$_H$	Record identifier CONTINUE
8230	2	`0007`$_H$ (7)	Record data size
8232	1	`01`$_H$	Unicode string option flags (16-bit-characters)
8233	2	`0048`$_H$	16-bit-character „H“
8235	2	`1234`$_H$	16-bit-character „Ø“
8237	2	`0049`$_H$	16-bit-character „I“

5.7 CRN

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	`005A`$_H$	`005A`$_H$	`005A`$_H$	`005A`$_H$	`005A`$_H$

This record stores the contents of an external cell or cell range. An external cell range has one row only. If a cell range spans over more than one row, several CRN records will be created. See →4.5 for details about external references.

Record CRN, BIFF3-BIFF8:

Offset	Size	Contents
0	1	Index to last column inside of the referenced sheet (`lc`)
1	1	Index to first column inside of the referenced sheet (`fc`)
2	2	Index to row inside of the referenced sheet
4	var.	List of `lc`-`fc`+1 cached values (→2.5)

5.8 DCONREF – Data Consolidation Reference

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
0051_H	0051_H	0051_H	0051_H	0051_H	0051_H

2do

5.9 DIMENSIONS

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
0000_H	0200_H	0200_H	0200_H	0200_H	0200_H

2do

5.10 EOF – End of File

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
$000A_H$	$000A_H$	$000A_H$	$000A_H$	$000A_H$	$000A_H$

This record has no content. It indicates the end of a record block with leading BOF record (→5.4). This could be the end of the workbook globals, a worksheet, a chart, etc.

5.11 EXTERNCOUNT

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
0016_H	0016_H	0016_H	0016_H	0016_H	---

This record contains the number of following EXTERNSHEET records. In BIFF8 this record is omitted because there occurs only one EXTERNSHEET record. See →4.5.1 for details about external references in BIFF2-BIFF4 and →4.5.2 for BIFF5/BIFF7.

Record EXTERNCOUNT, BIFF2-BIFF7:

Offset	Size	Contents
0	2	Number of following EXTERNSHEET records (→5.13)

5.12 EXTERNNAME

BIFF2	BIFF3	BIFF4	BIFF5	BIFF8
0023_H	0223_H	0223_H	0023_H	0023_H

This record contains the name of an external defined name, the name of an AddIn function, a DDE link item or an OLE object storage name (BIFF8).

• EXTERNNAME in BIFF2-BIFF7

The meaning of the name is dependent on the leading EXTERNSHEET record (→5.13). See →4.5.1 for details about external references in BIFF2-BIFF4 and →4.5.2 for BIFF5/BIFF7.

Record EXTERNNAME, BIFF2-BIFF7:

Offset	Size	Contents
0	var.	External name (byte string, 8-bit string length, →2.1)

If the record contains an item of a DDE link, a list with cached values will be appended to the string. These values are used as results for the DDE link. They are saved row by row for a DDE link that spans over several cells. Note: Only the results of the DDE link (the contents of the referenced cells) are stored, not the results of the complete formulas.

Record EXTERNNAME for DDE items, BIFF2-BIFF7:

Offset	Size	Contents
0	var.	DDE item (byte string, 8-bit string length, →2.1)
var.	1	Number of columns (nc). The value 0 means 256 columns.
var.	2	Number of rows (nr)
var.	var.	List of nc·nr cached values (→2.5)

• EXTERNNAME in BIFF8

In BIFF8 the record contains option flags which describe the type of the external name. So, this record must follow the correct SUPBOOK record (→5.35) and must contain the correct flags. See →4.5.3 for details about external references in BIFF8.

Record EXTERNNAME for external names and AddIn functions, BIFF8:

Offset	Size	Contents
0	2	Option flags (see below)
2	2	One-based sheet index. The value 0 means all sheets or AddIn function.
4	2	Not used
6	var.	External name or AddIn function name (Unicode string, 8-bit string length, →2.2)
var.	var.	For external names only: formula data (RPN token array, →3)

Record EXTERNNAME for DDE links, BIFF8:

Offset	Size	Contents
0	2	Option flags (see below)
2	4	Not used
6	var.	DDE item (Unicode string, 8-bit string length, →2.2)
var.	1	Number of columns decreased by 1 (nc)
var.	2	Number of rows decreased by 1 (nr)
var.	var.	List of (nc+1)·(nr+1) cached values (→2.5)

Record EXTERNNAME for OLE object links, BIFF8:

Offset	Size	Contents
0	2	Option flags (see below)
2	4	Storage identifier
6	3	01_H 00_H 27_H

Option flags:

Bit	Mask	Contents	
0	0001_H	0 = No BuiltIn name	1 = BuiltIn name
1	0002_H	0 = Manual DDE/OLE link	1 = Automatic DDE/OLE link
4	0010_H	0 = External name or DDE link	1 = OLE object link
14-5	$7FE0_H$	For DDE links only: clipboard format of last successful update	

5.13 EXTERNSHEET

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
`0017`$_H$	`0017`$_H$	`0017`$_H$	`0017`$_H$	`0017`$_H$	`0017`$_H$

• EXTERNSHEET in BIFF2-BIFF7

In the file format versions up to BIFF7 this record stores the name of an external document and a sheet name inside of this document. See →4.5.1 for details about external references in BIFF2-BIFF4 and →4.5.2 for BIFF5/BIFF7.

Record EXTERNSHEET, BIFF2-BIFF7:

Offset	Size	Contents
0	var.	Encoded document and sheet name (→2.6). Byte string, 8-bit string length (→2.1).

Attention: The string length field is decreased by 1, if the EXTERNSHEET stores a reference to one of the own sheets (first character is `03`$_H$). Example: The formula `=Sheet2!A1` contains a reference to an EXTERNSHEET record with the string „<`03`$_H$>Sheet2". The string consists of 7 characters but the string length field contains the value 6.

If a formula uses an AddIn function, a special EXTERNSHEET record will occur, followed by an EXTERNNAME record with the name of the function.

Record EXTERNSHEET for AddIn functions, BIFF2-BIFF7:

Offset	Size	Contents
0	2	`3401`$_H$ (`01`$_H$ `34`$_H$ = the byte string „#")

• EXTERNSHEET in BIFF8

In BIFF8 the record stores a list with indexes to SUPBOOK records (list of REF structures). See →4.5.3 for details about external references in BIFF8.

Record EXTERNSHEET, BIFF8:

Offset	Size	Contents		
0	2	Number of following REF structures (nm)		
2	6·nm	List of nm REF structures. Each REF contains the following data:		
		Offset	**Size**	**Contents**
		0	2	Index to SUPBOOK record
		2	2	Index to first SUPBOOK sheet
		4	2	Index to last SUPBOOK sheet

5.14 EXTSST – Extended SST

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	---	---	---	---	00FF$_H$

This record occurs in conjunction with the SST record (→5.33). It contains a hash table with stream offsets to the SST record to optimize string search operations. Excel does not shorten this record if strings are deleted from the shared string table, so the last part might contain invalid data. The stream indexes in this record divide the SST into hash buckets containing a constant number of strings. See →4.4 for more information about shared string tables.

Record EXTSST, BIFF8:

Offset	Size	Contents		
0	2	Number of strings in a hash bucket, this number is ⩾8		
2	var.	List of OFFSET structures. Each OFFSET contains the following data:		
		Offset	**Size**	**Contents**
		0	4	Absolute stream position of first string of this bucket
		4	2	Position of first string of this bucket inside of current record, including record header. This counter restarts at zero inside of CONTINUE records.
		6	2	Not used

5.15 FONT

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
0031$_H$	0231$_H$	0231$_H$	0031$_H$	0031$_H$	0031$_H$

This record contains information about an used font, including character formatting.

Some of the elements occur unchanged in every BIFF version. These elements are described in the following tables using a specific name for each element. In the description of the record structure the names are used to reference to these tables.

5.15.1 FONT substructures

• FONT_SCRIPT – Subscript or superscript (2 bytes), BIFF5-BIFF8

Value	Contents
0000$_H$	None
0001$_H$	Superscript
0002$_H$	Subscript

• FONT_UNDERLINE – Underline type (1 byte), BIFF5-BIFF8

Value	Contents
00$_H$	None
01$_H$	Single
02$_H$	Double
03$_H$	Single accounting
04$_H$	Double accounting

5.15.2 FONT record contents

Record FONT, BIFF2:

Offset	Size	Contents		
0	2	Height of the font (in 1/20 of a point)		
2	2	Option flags:		
		Bit	**Mask**	**Contents**
		0	0001$_H$	1 = Characters are bold
		1	0002$_H$	1 = Characters are italic
		2	0004$_H$	1 = Characters are underlined
		3	0008$_H$	1 = Characters are struck out
4	var.	Font name (byte string, 8-bit string length, →2.1)		

Record FONT, BIFF3-BIFF4:

Offset	Size	Contents		
0	2	Height of the font (in 1/20 of a point)		
2	2	Option flags:		
		Bit	**Mask**	**Contents**
		0	0001$_H$	1 = Characters are bold
		1	0002$_H$	1 = Characters are italic
		2	0004$_H$	1 = Characters are underlined
		3	0008$_H$	1 = Characters are struck out
4	2	Index into PALETTE record (→5.27)		
6	var.	Font name (byte string, 8-bit string length, →2.1)		

Record FONT, BIFF5-BIFF8:

Offset	Size	Contents		
0	2	Height of the font (in 1/20 of a point)		
2	2	Option flags:		
		Bit	**Mask**	**Contents**
		1	0002$_H$	1 = Characters are italic
		3	0008$_H$	1 = Characters are struck out
4	2	Index into PALETTE record (→5.27)		
6	2	Boldness (100-1000). Standard values are 0190$_H$ (400) for normal text and 02BC$_H$ (700) for bold text.		
8	2	FONT_SCRIPT – Subscript or superscript (see above)		
10	1	FONT_UNDERLINE – Underline type (see above)		
11	1	Font family...		
12	1	Character set...		
13	1	Not used		
14	var.	Font name:	BIFF5/BIFF7: Byte string, 8-bit string length (→2.1) BIFF8: Unicode string, 8-bit string length (→2.2)	

5.16 FORMAT

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
001E$_H$	001E$_H$	041E$_H$	041E$_H$	041E$_H$	041E$_H$

2do

5.17 FORMULA

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
0006H	0206H	0406H	0006H	0006H	0006H

This record contains the token array and the result of a formula cell.

• Record contents

Record FORMULA, BIFF2:

Offset	Size	Contents
0	2	Index to row
2	2	Index to column
4	3	Cell attributes (→2.9)
7	8	Result of the formula (IEEE floating-point value)
15	1	0 = Do not recalculate, 1 = Recalculate always
16	var.	Formula data (RPN token array, →3)

Record FORMULA, BIFF3-BIFF4:

Offset	Size	Contents		
0	2	Index to row		
2	2	Index to column		
4	2	Index to XF record (→5.37)		
6	8	Result of the formula. See below for details.		
14	2	Option flags:		
		Bit	Mask	Contents
		0	0001H	1 = Recalculate always
		1	0002H	1 = Calculate on open
16	var.	Formula data (RPN token array, →3)		

Record FORMULA, BIFF5-BIFF8:

Offset	Size	Contents		
0	2	Index to row		
2	2	Index to column		
4	2	Index to XF record (→5.37)		
6	8	Result of the formula. See below for details.		
14	2	Option flags:		
		Bit	Mask	Contents
		0	0001H	1 = Recalculate always
		1	0002H	1 = Calculate on open
		3	0008H	1 = Part of a shared formula
16	4	Not used		
20	var.	Formula data (RPN token array, →3)		

• Result of the formula

Dependent on the type of value the formula returns, the result field has the following format:
Result is a numeric value:

Offset	Size	Contents
0	8	IEEE floating-point value

Result is a string (the string itself follows in a STRING record, →5.34):

Offset	Size	Contents
0	1	00H (identifier for a string value)
1	5	Not used
6	2	FFFFH

Result is a boolean value:

Offset	Size	Contents
0	1	01H (identifier for a boolean value)
1	1	Not used
2	1	0 = FALSE, 1 = TRUE
3	3	Not used
6	2	FFFFH

Result is an error value:

Offset	Size	Contents
0	1	02H (identifier for an error value)
1	1	Not used
2	1	Error code (→2.4)
3	3	Not used
6	2	FFFFH

5.18 HLINK – Hyperlink

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	---	---	---	---	01B8H

In Excel, every cell can contain only one hyperlink. Therefore, the HLINK record refers to one cell address or a cell range where all cells contain the same hyperlink. Every hyperlink can contain a text mark and a description that is shown in the sheet instead of the real link. Text marks are appended behind a link, separated by the „#“ sign. Examples for text marks are www.xyz.org#table1 or C:\example.xls#Sheet1!A1.

Inside of this record strings are stored in several formats. Sometimes occurs the character count, otherwise the character array size (in 16-bit-character arrays the character count is half of the array size). Furthermore some strings are zero-terminated, others not. They are stored either as 16-bit-character arrays or as 8-bit-character arrays, independent of the characters.

5.18.1 Common record contents

Each HLINK record starts with the same data items and continues with special data related to the current type of hyperlink.

Record HLINK, BIFF8:

Offset	Size	Contents
0	2	Index to first row
2	2	Index to last row
4	2	Index to first column
6	2	Index to last column
8	20	Unknown byte sequence: `D0`$_H$ `C9`$_H$ `EA`$_H$ `79`$_H$ `F9`$_H$ `BA`$_H$ `CE`$_H$ `11`$_H$ `8C`$_H$ `82`$_H$ `00`$_H$ `AA`$_H$ `00`$_H$ `4B`$_H$ `A9`$_H$ `0B`$_H$ `02`$_H$ `00`$_H$ `00`$_H$ `00`$_H$
28	4	Option flags (see below)
[32]	4	(optional, see option flags) Character count of description text, including trailing zero word (`dl`)
[36]	2∙`dl`	(optional, see option flags) Character array of description text, no Unicode string header, always 16-bit-characters, zero-terminated
		Special data (→5.18.2 and following)
[var.]	4	(optional, see option flags) Character count of the text mark, including trailing zero word (`tl`)
[var.]	2∙`tl`	(optional, see option flags) Character array of the text mark without „#“ sign, no Unicode string header, always 16-bit-characters, zero-terminated

The special data parts in the following are described with relative offsets (starting again by zero). The real offset inside of the record data (without header) is either 32 (without description) or 36+2∙`dl` (with description).

• Option flags

The option flags specify the following content of the record.

Bit	Mask	Contents	
0	`00000001`$_H$	0 = No link extant	1 = File link or URL
1	`00000002`$_H$	0 = Relative file path	1 = Absolute path or URL
2 and 4	`00000014`$_H$	0 = No description	1 (both bits) = Description
3	`00000008`$_H$	0 = No text mark	1 = Text mark
8	`00000100`$_H$	0 = File link or URL	1 = Network path

5.18.2 Hyperlink to a common URL

These data fields occur for links which are not local files or files in the local network. The lower 9 bits of the option flags field must be `0.000x.xx11`$_2$ (x means optional, depending on hyperlink content). The byte sequence should be used to distinguish an URL from a file link.

Offset	Size	Contents
0	16	Unknown byte sequence, used as URL identifier: `E0`$_H$ `C9`$_H$ `EA`$_H$ `79`$_H$ `F9`$_H$ `BA`$_H$ `CE`$_H$ `11`$_H$ `8C`$_H$ `82`$_H$ `00`$_H$ `AA`$_H$ `00`$_H$ `4B`$_H$ `A9`$_H$ `0B`$_H$
16	4	Size of character array of the URL, including trailing zero word (`us`). There are `us`/2-1 characters in the following string.
20	`us`	Character array of the URL, no Unicode string header, always 16-bit-characters, zero-terminated

5.18.3 Hyperlink to a local file

These data fields are for links to files on local drives. The path of the file can be complete with drive letter (absolute) or relative to the location of the workbook. The lower 9 bits of the option flags field must be `0.000x.xxx1`$_2$. The byte sequence should be used to distinguish an URL from a file link.

Offset	Size	Contents
0	16	Unknown byte sequence, used as file link identifier: `03`$_H$ `03`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `C0`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `46`$_H$
16	2	Directory up-level count. Each leading „..\" in the file link is deleted and inceases this counter.
18	4	Character count of the shortened file path and name, including trailing zero byte (`sl`)
22	`sl`	Character array of the shortened file path and name in 8.3-DOS-format. This field can be filled with a long file name too. No Unicode string header, always 8-bit-characters, zero-terminated.
22+`sl`	24	Unknown byte sequence: `FF`$_H$ `FF`$_H$ `AD`$_H$ `DE`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$ `00`$_H$
46+`sl`	4	Size of the following file link field including string length field and additional data field (`sz`). If `sz` is zero, nothing will follow (except a text mark).
[50+`sl`]	4	(optional) Size of character array of the extended file path and name (`xl`). There are `xl`/2 characters in the following string.
[54+`sl`]	2	(optional) Unknown byte sequence: `03`$_H$ `00`$_H$
[56+`sl`]	`xl`	(optional) Character array of the extended file path and name (`xl`), no Unicode string header, always 16-bit-characters, not zero-terminated

5.18.4 Hyperlink to a file located in the local network

These data fields are for links to files located in the local network. The lower 9 bits of the option flags field must be `1.000x.xx11`$_2$.

Offset	Size	Contents
0	4	Character count of the network path and file name, including trailing zero word (`fl`)
4	2·`fl`	Character array of the network path and file name, no Unicode string header, always 16-bit-characters, zero-terminated.

5.18.5 Hyperlink to a place in the current workbook

In this case only the text mark field is present (optional with description). Example: The URL „#Sheet2!B1:C2" refers to the given range in the current workbook. The lower 9 bits of the option flags field must be `0.000x.1x00`$_2$.

5.19 INTEGER

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
0002$_H$	---	---	---	---	---

This record represents a cell that contains an unsigned 16-bit-integer value. If a value cannot be stored as a 16-bit-integer, a NUMBER record (→5.26) must be written. From BIFF3 on this record is replaced by the RK record (→5.29).

Record INTEGER, BIFF2:

Offset	Size	Contents
0	2	Index to row
2	2	Index to column
4	3	Cell attributes (→2.9)
7	2	Unsigned 16-bit-integer value

5.20 IXFE – Index to XF

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
0044$_H$	---	---	---	---	---

This record occurs in front of every cell record (for instance BLANK, INTEGER, NUMBER, LABEL, FORMULA) that references to an XF record (→5.37) with an index greater than 62. The XF index field of the cell record consists only of 6 bits. The maximum value 63 is used to indicate a preceding IXFE record containing the real XF index. See →2.9 for more details.

Record IXFE, BIFF2:

Offset	Size	Contents
0	2	Index to XF record (→5.37)

5.21 LABEL

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
0004$_H$	0204$_H$	0204$_H$	0204$_H$	0204$_H$	---

This record represents a cell that contains a string. In BIFF8 it is replaced by the LABELSST record (→5.22).

Record LABEL, BIFF2:

Offset	Size	Contents
0	2	Index to row
2	2	Index to column
4	3	Cell attributes (→2.9)
7	var.	Byte string, 8-bit string length (→2.1)

Record LABEL, BIFF3-BIFF7:

Offset	Size	Contents
0	2	Index to row
2	2	Index to column
4	2	Index to XF record (→5.37)
6	var.	Byte string, 16-bit string length (→2.1)

5.22 LABELSST

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	---	---	---	---	00FD$_H$

This record represents a cell that contains a string. It replaces the LABEL record (→5.21) used in BIFF2-BIFF7. See →4.4 for more information about shared string tables.

Record LABELSST, BIFF8:

Offset	Size	Contents
0	2	Index to row
2	2	Index to column
4	2	Index to XF record (→5.37)
6	4	Index into SST record (→5.33)

5.23 MULBLANK – Multiple BLANK

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	---	---	00BE$_H$	00BE$_H$	00BE$_H$

This record represents a cell range of empty cells. All cells are located in the same row.

Record MULBLANK, BIFF5-BIFF8:

Offset	Size	Contents
0	2	Index to row
2	2	Index to first column (`fc`)
4	2·(`lc`-`fc`+1)	Array of `lc`-`fc`+1 16-bit-indexes to XF records (→5.37)
var.	2	Index to last column (`lc`)

5.24 MULRK – Multiple RK

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	---	---	00BD$_H$	00BD$_H$	00BD$_H$

This record represents a cell range containing RK value cells. All cells are located in the same row.

Record MULRK, BIFF5-BIFF8:

Offset	Size	Contents
0	2	Index to row
2	2	Index to first column (`fc`)
4	6·(`lc`-`fc`+1)	Array of `lc`-`fc`+1 XF/RK structures. Each XF/RK contains:

Offset	Size	Contents
0	2	Index to XF record (→5.37)
2	4	RK value (→2.3)

Offset	Size	Contents
var.	2	Index to last column (`lc`)

5.25 NAME

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
0018_H	0218_H	0218_H	0018_H	0018_H	0018_H

This record contains the name and the token array of an internal defined name.
Record NAME, BIFF2:

Offset	Size	Contents		
0	1	Option flags:		
		Bit	**Mask**	**Contents**
		1	02_H	1 = Function macro or command macro
		2	04_H	1 = Complex function (array formula or user defined)
1	1	If name is function macro or command macro (see option flags above): 01_H = Function macro, 02_H = Command macro		
2	1	Keyboard shortcut		
3	1	Length of the name (character count) (ln)		
4	1	Size of the formula data (RPN token array) (sz)		
5	ln	Character array of the name		
5+ln	sz	Formula data (RPN token array without size field, →3)		
5+ln+sz	1	Duplicate of the formula data size field (sz)		

Record NAME, BIFF3-BIFF4:

Offset	Size	Contents		
0	2	Option flags:		
		Bit	**Mask**	**Contents**
		0	0001_H	1 = Name is hidden
		1	0002_H	1 = Name is a function
		2	0004_H	1 = Name is a command
		3	0008_H	1 = Function macro or command macro
		4	0010_H	1 = Complex function (array formula or user defined)
		5	0020_H	1 = Built-in name (see table below)
		11-6	$0FC0_H$	BIFF3: Not used; BIFF4: Index to function group
2	1	Keyboard shortcut		
3	1	Length of the name (character count) (ln)		
4	2	Size of the formula data (RPN token array) (sz)		
6	ln	Character array of the name		
6+ln	sz	Formula data (RPN token array without size field, →3)		

Record NAME, BIFF5/BIFF7:

Offset	Size	Contents
0	2	Option flags:
		Bit / **Mask** / **Contents**
		11-0 / `0FFF`$_H$ / Equal to BIFF4 (see table above)
		12 / `1000`$_H$ / 1 = Name contains binary data
2	1	Keyboard shortcut
3	1	Length of the name (character count) (`ln`)
4	2	Size of the formula data (RPN token array) (`sz`)
6	2	Unused
8	2	0 = Global name, otherwise index to sheet (one-based)
10	1	Length of menu text (character count) (`lm`)
11	1	Length of description text (character count) (`ld`)
12	1	Length of help topic text (character count) (`lh`)
13	1	Length of status bar text (character count) (`ls`)
14	`ln`	Character array of the name
14+`ln`	`sz`	Formula data (RPN token array without size field, →3)
14+`ln`+`sz`	`lm`	Character array of menu text
var.	`ld`	Character array of description text
var.	`lh`	Character array of help topic text
var.	`ls`	Character array of status bar text

Record NAME, BIFF8:

Offset	Size	Contents
0	2	Option flags: Equal to BIFF5/BIFF7 (see table above)
2	1	Keyboard shortcut
3	1	Length of the name (character count)
4	2	Size of the formula data (RPN token array) (`sz`)
6	2	Unused
8	2	0 = Global name, otherwise index to sheet (one-based)
10	1	Length of menu text (character count)
11	1	Length of description text (character count)
12	1	Length of help topic text (character count)
13	1	Length of status bar text (character count)
14	var.	Name (Unicode string without length field, →2.2)
var.	`sz`	Formula data (RPN token array without size field, →3)
var.	var.	Menu text (Unicode string without length field, →2.2)
var.	var.	Description text (Unicode string without length field, →2.2)
var.	var.	Help topic text (Unicode string without length field, →2.2)
var.	var.	Status bar text (Unicode string without length field, →2.2)

• Built-in names

From BIFF3 on only an index to a built-in names is stored. If bit 5 of the option flags field is set, the name string contains only one character with this index.

Built-in index	Built-In name
00_H	Consolidate_Area
01_H	Auto_Open
02_H	Auto_Close
03_H	Extract
04_H	Database
05_H	Criteria
06_H	Print_Area
07_H	Pint_Titles
08_H	Recorder
09_H	Data_Form
$0A_H$	Auto_Activate
$0B_H$	Auto_Deactivate
$0C_H$	Sheet_Title
$0D_H$	BIFF3-BIFF4: Not used; BIFF5-BIFF8: Autofilter

5.26 NUMBER

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
0003_H	0203_H	0203_H	0203_H	0203_H	0203_H

This record represents a cell that contains a floating-point value.

Record NUMBER, BIFF2:

Offset	Size	Contents
0	2	Index to row
2	2	Index to column
4	3	Cell attributes (→2.9)
7	8	IEEE floating-point value

Record NUMBER, BIFF3-BIFF8:

Offset	Size	Contents
0	2	Index to row
2	2	Index to column
4	2	Index to XF record (→5.37)
6	8	IEEE floating-point value

5.27 PALETTE

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	0092_H	0092_H	0092_H	0092_H	0092_H

This record contains the definition of all colors available for cell and object formatting.
Record PALETTE, BIFF3-BIFF8:

Offset	Size	Contents
0	2	Number of following colors (nm)
2	4·nm	List of nm colors. Each color contains:

Offset	Size	Contents
0	1	Red component of the color
1	1	Green component of the color
2	1	Blue component of the color
3	1	Not used

5.28 PASSWORD

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
0013_H	0013_H	0013_H	0013_H	0013_H	0013_H

This record stores a 16-bit hash value for a sheet or workbook protection password.

Offset	Size	Contents
0	2	16-bit hash value of the password

This is the algorithm to create the hash value from a given password:

- The ASCII values of all characters are rotated left with a number of digits depending on the character position (first character is rotated left 1 bit, second character 2 bits, and so on). There is a space of 15 bits available for rotation (bit 15 jumps to bit 0, bit 16 jumps to bit 1 and so on).
- All rotated characters are combined using XOR operation.
- The number of characters is added using XOR operation.
- The constant $\texttt{CE4B}_H$ is added using XOR operation.

Example: The password is „abcdefghij“ (10 characters).

Character	ASCII	Shifted	Rotated
a	61_H	$000000C2_H$	$00C2_H$
b	62_H	00000188_H	0188_H
c	63_H	00000318_H	0318_H
d	64_H	00000640_H	0640_H
e	65_H	$00000CA0_H$	$0CA0_H$
f	66_H	00001980_H	1980_H
g	67_H	00003380_H	3380_H
h	68_H	00006800_H	6800_H
i	69_H	$0000D200_H$	5201_H
j	$6A_H$	$0001A800_H$	2803_H

All the rotated values and the number of characters $\texttt{000A}_H$ and the constant $\texttt{CE4B}_H$ result in the hash value $\texttt{FEF1}_H$.

5.29 RK

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	027E$_H$	027E$_H$	027E$_H$	027E$_H$	027E$_H$

This record represents a cell that contains an RK value (encoded integer or floating-point value). If a floating-point value cannot be encoded to an RK value, a NUMBER record (→5.26) must be written. This record replaces the record INTEGER (→5.19) written in BIFF2.

Record RK, BIFF3-BIFF8:

Offset	Size	Contents
0	2	Index to row
2	2	Index to column
4	2	Index to XF record (→5.37)
6	4	RK value (→2.3)

5.30 SCREENTIP

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	---	---	---	---	0800$_H$

This record contains the cell range and text for a screen tip. It occurs in conjunction with the HLINK record for hyperlinks (→5.18).

Record SCREENTIP, BIFF8:

Offset	Size	Contents
0	2	0800$_H$ (repeated record ID)
2	2	Index to first row
4	2	Index to last row
6	2	Index to first column
8	2	Index to last column
10	var.	Character array of the screen tip, no Unicode string header, always 16-bit-characters, zero-terminated

5.31 SHEETHDR

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	---	008F$_H$	---	---	---

This record occurs only in BIFF4 workbook files. It precedes a substream for a sheet (delimited by a BOF and a EOF record) and contains the byte length of the substream and the sheet name. Adding the substream length to the stream position of the following BOF record gives the position of the next SHEETHDR record. See →4.2 for details about the BIFF4 workbook stream.

Record SHEETHDR, BIFF4:

Offset	Size	Contents
0	4	Byte length of the following sheet substream
4	var.	Name of the sheet (byte string, 8-bit string length, →2.1)

5.32 SHEETSOFFSET

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	---	008E$_H$	---	---	---

This record occurs only in BIFF4 workbook files. It is located in the workbook globals section and contains the stream position of the first SHEETHDR record (→5.31). See →4.2 for details about the BIFF4 workbook stream.

Record SHEETSOFFSET, BIFF4:

Offset	Size	Contents
0	4	Stream position of the first SHEETHDR record (→5.31)

5.33 SST – Shared String Table

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	---	---	---	---	00FC$_H$

This record contains a list of all strings used anywhere in the workbook. Each string occurs only one time. The workbook uses indexes into the list to reference to strings. See →4.4 for more information.

Record SST, BIFF8:

Offset	Size	Contents
0	4	Total number of strings in the workbook (see below)
4	4	Number of following strings (nm)
8	var.	List of nm Unicode strings, 16-bit string length (→2.2)

The first field of the SST record counts the total occurrence of strings in the workbook. For instance, the string „AAA“ is used 3 times and the string „BBB“ is used 2 times. The first field contains 5 and the second field contains 2, followed by the two strings.

5.34 STRING

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
0007$_H$	0207$_H$	0207$_H$	0207$_H$	0207$_H$	0207$_H$

This record stores the result of a string formula. It occurs directly after a string formula (→5.17).

Record STRING, BIFF2:

Offset	Size	Contents
0	var.	Byte string, 8-bit string length (→2.1)

Record STRING, BIFF3-BIFF7:

Offset	Size	Contents
0	var.	Byte string, 16-bit string length (→2.1)

In BIFF8 files the whole record is omitted, if the result is an empty string.

Record STRING, BIFF8:

Offset	Size	Contents
0	var.	Unicode string with at least 1 character, 16-bit string length (→2.2)

5.35 SUPBOOK – External Workbook

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	---	---	---	---	01AE$_H$

This record mainly stores the name of an external document and a list of sheet names inside of this document. Furthermore it is used to store names of documents for DDE and OLE object links or to indicate an internal 3D reference or an AddIn function. See →4.5.3 for details about external references in BIFF8.

5.35.1 External references

A SUPBOOK record for external references stores the name of the document and a list of sheet names.

Record SUPBOOK for external references, BIFF8:

Offset	Size	Contents
0	2	Number of sheet names (nm)
2	var.	Encoded document name without sheet name (→2.6.1). Unicode string, 16-bit string length (→2.2).
var.	var.	List of nm sheet names (Unicode strings with 16-bit string length, →2.2)

5.35.2 Internal references

In each file occurs a SUPBOOK that is used for internal 3D references. It stores the number of sheets of the own document.

Record SUPBOOK for 3D references, BIFF8:

Offset	Size	Contents
0	2	Number of sheets in this document
2	2	0401$_H$

5.35.3 AddIn functions

AddIn function names are stored in EXTERNNAME records following this SUPBOOK record.

Record SUPBOOK for AddIn functions, BIFF8:

Offset	Size	Contents
0	2	0001$_H$
2	2	3A01$_H$

5.35.4 DDE links, OLE object links

The SUPBOOK record of a DDE link or an OLE object link contains the name of the server application (DDE) or the class name (OLE) and the name of a source document. These names are encoded in one string.

Record SUPBOOK for DDE links and OLE object links, BIFF8:

Offset	Size	Contents
0	2	0000$_H$
2	var.	Encoded source document name (→2.6.2). Unicode string, 16-bit string length (→2.2).

5.36 XCT – CRN Count

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
---	`0059`H	`0059`H	`0059`H	`0059`H	`0059`H

This record stores the number of immediately following CRN records. These records are used to store the cell contents of external references. See →4.5 for details about of external references.

Record XCT, BIFF3-BIFF7:

Offset	Size	Contents
0	2	Number of following CRN records

Record XCT, BIFF8:

Offset	Size	Contents
0	2	Number of following CRN records
2	2	Index to sheet table of the involved SUPBOOK record (→5.35)

5.37 XF – Extended Format

BIFF2	BIFF3	BIFF4	BIFF5	BIFF7	BIFF8
`0043`H	`0243`H	`0443`H	`00E0`H	`00E0`H	`00E0`H

This record contains formatting information for cells, rows, columns or styles.

From BIFF3 on, some of the elements occur unchanged in every BIFF version. These elements are described in the following using a specific name for each element. In the description of the record structure the names are used to reference to these tables.

5.37.1 XF substructures

• XF_TYPE_PROT – XF type and cell protection (3 bits), BIFF3-BIFF8

These 3 bits are part of a specific data byte.

Bit	Mask	Contents
0	`01`H	1 = Cell is locked
1	`02`H	1 = Formula is hidden
2	`04`H	0 = Cell XF; 1 = Style XF

• XF_USED_ATTRIB – Attributes used from parent style XF (1 byte), BIFF3-BIFF8

In this byte, each bit describes the validity of a specific attribute. In cell XFs an unset bit means the attribute of the parent style XF is used, a set bit means the attribute of this XF is used. In style XFs an unset bit means the attribute setting is valid, a set bit means the attribute should be ignored.

Bit	Mask	Contents
2	`04`H	Flag for number format
3	`08`H	Flag for font
4	`10`H	Flag for alignment, text wrap and rotation
5	`20`H	Flag for border lines
6	`40`H	Flag for background area style
7	`80`H	Flag for cell protection (cell locked and formula hidden)

• XF_HOR_ALIGN – Horizontal alignment (3 bits), BIFF2-BIFF8

The horizontal alignment consists of 3 bits and is part of a specific data byte.

Value	Horizontal alignment
00_H	General
01_H	Left
02_H	Centered
03_H	Right
04_H	Filled
05_H	Justified (BIFF3-BIFF8)
06_H	Centered across selection (BIFF3-BIFF8)

• XF_VERT_ALIGN – Vertical alignment (2 bits), BIFF4-BIFF8

The vertical alignment consists of 2 bits and is part of a specific data byte. Vertical alignment is not available in BIFF2 and BIFF3.

Value	Vertical alignment
00_H	Left
01_H	Centered
02_H	Right
03_H	Justified (BIFF5-BIFF8)

• XF_ORIENTATION – Text orientation (2 bits), BIFF4-BIFF7

In the BIFF versions BIFF4-BIFF7, text can be rotated in steps of 90-degrees or stacked. The orientation mode consists of 2 bits and is part of a specific data byte. In BIFF8 a rotation angle occurs instead of these flags.

Value	Text orientation
00_H	Not rotated
01_H	Letters are stacked top-to-bottom, but not rotated
02_H	Text is rotated 90 degrees counterclockwise
03_H	Text is rotated 90 degrees clockwise

• XF_ROTATION – Text rotation angle (1 byte), BIFF8

Value	Text rotation
0	Not rotated
1-90	1 deg. - 90 deg. counterclockwise
91-180	1 deg. - 90 deg. clockwise
255	Letters are stacked top-to-bottom, but not rotated

• XF_BORDER_34 – Cell border style (4 bytes), BIFF3-BIFF4

Cell borders contain a line style and a line color for each line of the border.

Bit	Mask	Contents
2-0	00000007H	Top line style (→2.7)
7-3	000000F8H	Index into PALETTE record for top line color (→5.27)
10-8	00000700H	Left line style
15-11	0000F800H	Index into PALETTE record for left line color
18-16	00070000H	Bottom line style
23-19	00F80000H	Index into PALETTE record for bottom line color
26-24	07000000H	Right line style
31-27	F8000000H	Index into PALETTE record for right line color

• XF_AREA_34 – Cell background area style (2 bytes), BIFF3-BIFF4

A cell background area style contains an area pattern and a foreground and background color.

Bit	Mask	Contents
5-0	003FH	Fill pattern (→2.8)
10-6	07C0H	Index into PALETTE record for pattern foreground (→5.27)
15-11	F800H	Index into PALETTE record for pattern background

5.37.2 XF record contents

Record XF, BIFF2:

Offset	Size	Contents		
0	1	Index to FONT record (→5.15)		
1	1	Not used		
2	1	**Bit**	**Mask**	**Contents**
		5-0	3FH	Index to FORMAT record (→5.16)
		6	40H	1 = Cell is locked
		7	80H	1 = Formula is hidden
3	1	**Bit**	**Mask**	**Contents**
		2-0	07H	XF_HOR_ALIGN – Horizontal alignment (see above)
		3	08H	1 = Cell has left black border
		4	10H	1 = Cell has right black border
		5	20H	1 = Cell has top black border
		6	40H	1 = Cell has bottom black border
		7	80H	1 = Cell has shaded background

Record XF, BIFF3:

Offset	Size	Contents		
0	1	Index to FONT record (➜5.15)		
1	1	Index to FORMAT record (➜5.16)		
2	1	XF_TYPE_PROT – XF type and cell protection (see above)		
3	1	XF_USED_ATTRIB – Used attributes (see above)		
4	2	**Bit**	**Mask**	**Contents**
		2-0	0007_H	XF_HOR_ALIGN – Horizontal alignment (see above)
		3	0008_H	1 = Text is wrapped at right border
		15-4	$FFF0_H$	Index to parent style XF (always FFF_H in style XFs)
6	2	XF_AREA_34 – Cell background area (see above)		
8	4	XF_BORDER_34 – Cell border lines (see above)		

Record XF, BIFF4:

Offset	Size	Contents		
0	1	Index to FONT record (➜5.15)		
1	1	Index to FORMAT record (➜5.16)		
2	2	**Bit**	**Mask**	**Contents**
		2-0	0007_H	XF_TYPE_PROT – XF type, cell protection (see above)
		15-4	$FFF0_H$	Index to parent style XF (always FFF_H in style XFs)
4	1	**Bit**	**Mask**	**Contents**
		2-0	07_H	XF_HOR_ALIGN – Horizontal alignment (see above)
		3	08_H	1 = Text is wrapped at right border
		5-4	30_H	XF_VERT_ALIGN – Vertical alignment (see above)
		7-6	$C0_H$	XF_ORIENTATION – Text orientation (see above)
5	1	XF_USED_ATTRIB – Used attributes (see above)		
6	2	XF_AREA_34 – Cell background area (see above)		
8	4	XF_BORDER_34 – Cell border lines (see above)		

Record XF, BIFF5/BIFF7:

Offset	Size	Contents		
0	2	Index to FONT record (➜5.15)		
2	2	Index to FORMAT record (➜5.16)		
4	2	**Bit**	**Mask**	**Contents**
		2-0	0007$_H$	XF_TYPE_PROT – XF type, cell protection (see above)
		15-4	FFF0$_H$	Index to parent style XF (always FFF$_H$ in style XFs)
6	1	**Bit**	**Mask**	**Contents**
		2-0	07$_H$	XF_HOR_ALIGN – Horizontal alignment (see above)
		3	08$_H$	1 = Text is wrapped at right border
		5-4	30$_H$	XF_VERT_ALIGN – Vertical alignment (see above)
7	1	**Bit**	**Mask**	**Contents**
		1-0	03$_H$	XF_ORIENTATION – Text orientation (see above)
		7-2	FC$_H$	XF_USED_ATTRIB – Used attributes (see above)
8	4	Cell border lines and background area:		
		Bit	**Mask**	**Contents**
		6-0	0000007F$_H$	Index into PALETTE for pattern foreground
		13-7	00003F8$_H$	Index into PALETTE for pattern background
		21-16	003F0000$_H$	Fill pattern (➜2.8)
		24-22	01C00000$_H$	Bottom line style (➜2.7)
		31-25	FE000000$_H$	Index into PALETTE for bottom line color
12	4	**Bit**	**Mask**	**Contents**
		2-0	00000007$_H$	Top line style (➜2.7)
		5-3	00000038$_H$	Left line style
		8-6	000001C0$_H$	Right line style
		15-9	0000FE00$_H$	Index into PALETTE for top line color
		22-16	007F0000$_H$	Index into PALETTE for left line color
		29-23	3F800000$_H$	Index into PALETTE for right line color

Record XF, BIFF8:

Offset	Size	Contents		
0	2	Index to FONT record (➜5.15)		
2	2	Index to FORMAT record (➜5.16)		
4	2	**Bit**	**Mask**	**Contents**
		2-0	0007$_H$	XF_TYPE_PROT – XF type, cell protection (see above)
		15-4	FFF0$_H$	Index to parent style XF (always FFF$_H$ in style XFs)
6	1	**Bit**	**Mask**	**Contents**
		2-0	07$_H$	XF_HOR_ALIGN – Horizontal alignment (see above)
		3	08$_H$	1 = Text is wrapped at right border
		5-4	30$_H$	XF_VERT_ALIGN – Vertical alignment (see above)
7	1	XF_ROTATION: Text rotation angle (see above)		
8	1	**Bit**	**Mask**	**Contents**
		3-0	0F$_H$	Indent level
		4	10$_H$	1 = Shrink content to fit into cell
		5	20$_H$	1 = Cell is part of a merged range
9	1	**Bit**	**Mask**	**Contents**
		7-2	FC$_H$	XF_USED_ATTRIB – Used attributes (see above)
10	4	Cell border lines and background area:		
		Bit	**Mask**	**Contents**
		3-0	0000000F$_H$	Left line style (➜2.7)
		7-4	000000F0$_H$	Right line style
		11-8	00000F00$_H$	Top line style
		15-12	0000F000$_H$	Bottom line style
		22-16	007F0000$_H$	Index into PALETTE for left line color
		29-23	3F800000$_H$	Index into PALETTE for right line color
		30	40000000$_H$	1 = Diagonal line from top left to right bottom
		31	80000000$_H$	1 = Diagonal line from bottom left to right top
14	4	**Bit**	**Mask**	**Contents**
		6-0	0000007F$_H$	Index into PALETTE for top line color
		13-7	00003F80$_H$	Index into PALETTE for bottom line color
		20-14	001FC000$_H$	Index into PALETTE for diagonal line color
		24-21	01E00000$_H$	Diagonal line style (➜2.7)
		31-26	FC000000$_H$	Fill pattern (➜2.8)
18	2	**Bit**	**Mask**	**Contents**
		6-0	007F$_H$	Index into PALETTE for pattern foreground
		13-7	3F80$_H$	Index into PALETTE for pattern background

6 Drawing Objects, Escher Layer

2do

7 Charts

2do

8 PivotTables

2do

9 Change Tracking

2do

TWAIN Specification

Version 1.8

This document has been ratified by the TWAIN Working Group Committee as of October 22, 1998

TWAIN
Linking Images With Applications

Acknowledgments

The TWAIN Working Group acknowledges the following individuals and their respective companies for their contributions to this document. Their hard work in definition, design, editing, proofreading and discussing the evolution of the document have been invaluable.

Canon Information Systems
Ron Vogel
Engineering Manager

Eastman Kodak
Mark McLaughlin
Senior Software Engineer

Fujitsu Computer Products of America
Kanghoon Lee
Lead Engineer

Hewlett-Packard
Chuck Mayne
Software Development Engineer

Intel
Mannan Mohammed
Software Manager, Digital Imaging & Video Division

JFL Peripheral Solutions Inc.
Jon Harju
Director of Engineering

We also would like to thank the TWAIN Working Group Technical Committee for their opinions and contributions.

TWAIN
Linking Images With Applications

Contents

1

Introduction

Chapter Contents

The Need for Consistency

With the introduction of scanners, digital cameras, and other image acquisition devices, users eagerly discovered the value of incorporating images into their documents and other work. However, supporting the display and manipulation of this raster data placed a high cost on application developers. They needed to create user interfaces and build in device control for the wide assortment of available image devices. Once their application was prepared to support a given device, they faced the discouraging reality that devices continue to be upgraded with new capabilities and features. Application developers found themselves continually revising their product to stay current.

Developers of both the image acquisition devices and the software applications recognized the need for a standard communication between the image devices and the applications. A standard would benefit both groups as well as the users of their products. It would allow the device vendors' products to be accessed by more applications and application vendors could access data from those devices without concern for which type of device, or particular device, provided it. TWAIN was developed because of this need for consistency and simplification.

The Elements of TWAIN

TWAIN defines a standard software protocol and API (application programming interface) for communication between software applications and image acquisition devices (the source of the data).

The three key elements in TWAIN are:

- **The application software -** An application must be modified to use TWAIN.
- **The Source Manager software -** This software manages the interactions between the application and the Source. This code is provided in the TWAIN Developer's Toolkit and should be shipped for free with each TWAIN application and Source.
- **The Source software -** This software controls the image acquisition device and is written by the device developer to comply with TWAIN specifications. Traditional device drivers are now included with the Source software and do not need to be shipped by applications.

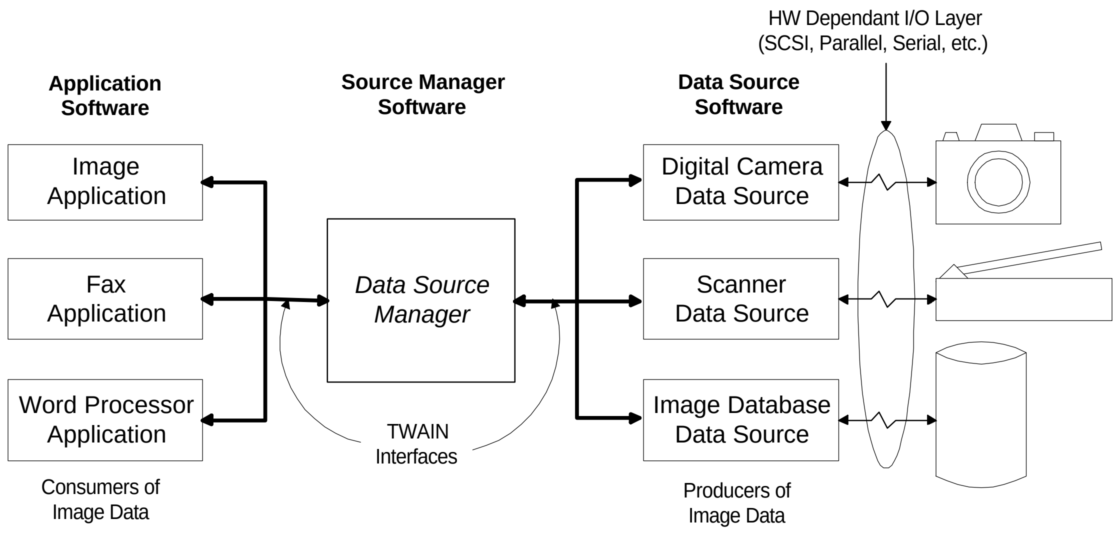


Figure 1-1. TWAIN Elements

The Benefits of Using TWAIN

For the Application Developer

- Allows you to offer users of your application a simple way to incorporate images from any compatible raster device without leaving your application.
- Saves time and dollars. If you currently provide low-level device drivers for scanners, etc., you no longer need to write, support, or ship these drivers. The TWAIN-compliant image acquisition devices will provide Source software modules that eliminate the need for you to create and ship device drivers.
- Permits your application to access data from any TWAIN-compliant image peripheral simply by modifying your application code once using the high-level TWAIN application programming interface. No customization by product is necessary. TWAIN image peripherals can include desktop scanners, hand scanners, digital cameras, frame grabbers, image databases, or any other raster image source that complies to the TWAIN protocol and API .
- Allows you to determine the features and capabilities that an image acquisition device can provide. Your application can then restrict the Source to offer only those capabilities that are compatible with your application’s needs and abilities.
- Eliminates the need for your application to provide a user interface to control the image acquisition process. There is a software user interface module shipped with every TWAIN-compliant Source device to handle that process. Of course, you may provide your own user interface for acquisition, if desired.

For the Source Developer

- Increases the use and support of your product. More applications will become image consumers as a result of the ease of implementation and breadth of device integration that TWAIN provides.
- Allows you to provide a proprietary user interface for your device. This lets you present the newest features to the user without waiting for the applications to incorporate them into their interfaces.
- Saves money by reducing your implementation costs. Rather than create and support various versions of your device control software to integrate with various applications, you create just a single TWAIN-compliant Source.

For the End User

- Gives users a simple way to incorporate images into their documents. They can access the image in fewer steps because they never need to leave your application.

The Creation of TWAIN

TWAIN was created by a small group of software and hardware companies in response to the need for a proposed specification for the imaging industry. The Working Group's goal was to provide an open, multi-platform solution to interconnect the needs of raster input devices with application software. The original Working Group was comprised of representatives from five companies: Aldus, Caere, Eastman Kodak, Hewlett-Packard, and Logitech. Three other companies, Adobe, Howtek, and Software Architects also contributed significantly.

The design of TWAIN began in January, 1991. Review of the original TWAIN Developer's Toolkit occurred from April, 1991 through January, 1992. The original Toolkit was reviewed by the TWAIN Coalition. The Coalition includes approximately 300 individuals representing 200 companies who continue to influence and guide the future direction of TWAIN.

The current version of TWAIN was written by the current 11 members of the TWAIN Working Group. The members include: Adobe, Canon, Eastman Kodak Company, Fujitsu Computer Products of America, Genoa Technology, Inc., Hewlett-Packard Company, Intel Corporation, J.F.L. Peripherals, Kofax Image Products, Ricoh Corporation, and Xerox.

In May, 1998, an agreement was announced between Microsoft and the TWAIN Working Group which provided for the inclusion of the TWAIN Data Source Manager in Microsoft Windows 98 and Microsoft Windows NT 5.0.

During the creation of TWAIN, the following architecture objectives were adhered to:

- **Ease of Adoption -** Allow an application vendor to make their application TWAIN-compliant with a reasonable amount of development and testing effort. The basic features of TWAIN should be implemented just by making modest changes to the application. To take advantage of a more complete set of functionality and control capabilities, more development effort should be anticipated.
- **Extensibility -** The architecture must include the flexibility to embrace multiple windowing environments spanning various host platforms (Macintosh, Microsoft Windows, Motif, etc.) and facilitate the exchange of various data types between Source devices and destination applications. Currently, only the raster image data type is supported but suggestions for future extensions include text, facsimile, vector graphics, and others.
- **Integration -** Key elements of the TWAIN implementation "belong" in the operating system. The agreement between Microsoft and the TWAIN Working Group indicates that this integration into the operating system is beginning. TWAIN must be implemented to encourage backward compatibility (extensibility) and smooth migration into the operating system. An implementation that minimizes the use of platform-specific mechanisms will have enhanced longevity and adoptability.
- **Easy Application <-> Source Interconnect -** A straight-forward Source identification and selection mechanism will be supplied. The application will drive this mechanism through a simple API. This mechanism will also establish the data and control links between the application and Source. It will support capability and configuration communication and negotiation between the application and Source.
- **Encapsulated Human Interface -** A device-native user interface will be required in each Source. The application can optionally override this native user interface while still using the Source to control the physical device.

2

Technical Overview

The TWAIN protocol and API are easiest to understand when you see the overall picture. This chapter describes:

Chapter Contents

The TWAIN Architecture

The transfer of data is made possible by three software elements that work together in TWAIN: the application, the Source Manager, and the Source.

These elements use the architecture of TWAIN to communicate. The TWAIN architecture consists of four layers:

- Application
- Protocol
- Acquisition
- Device

The TWAIN software elements occupy the layers as illustrated below. Each layer is described in the sections that follow.

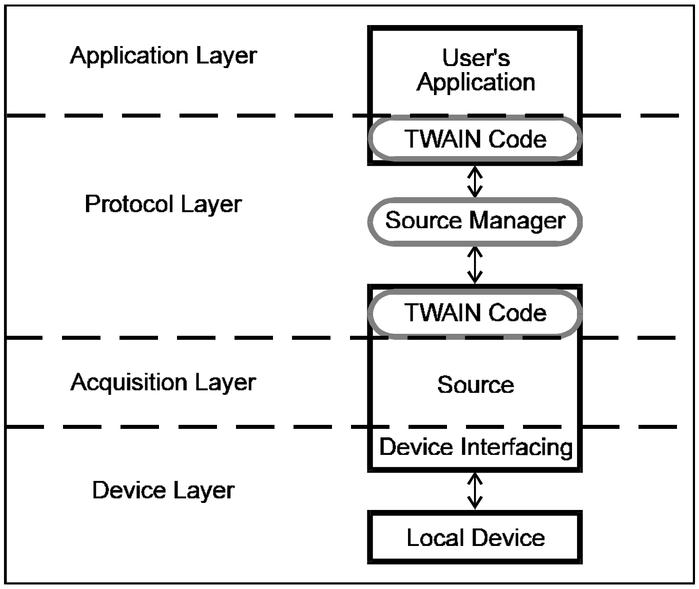


Figure 2-1. TWAIN Software Elements

Application

The user's software application executes in this layer.

TWAIN describes user interface guidelines for the application developer regarding how users access TWAIN functionality and how a particular Source is selected.

TWAIN is not concerned with how the application is implemented. TWAIN has no effect on any inter-application communication scheme that the application may use.

Protocol

The protocol is the "language" spoken and syntax used by TWAIN. It implements precise instructions and communications required for the transfer of data.

The protocol layer includes:

- The portion of application software that provides the interface between the application and TWAIN
- The TWAIN Source Manager provided by TWAIN
- The software included with the Source device to receive instructions from the Source Manager and transfer back data and Return Codes

The contents of the protocol layer are discussed in more detail in a following section called "Communication between the Elements of TWAIN."

Acquisition

Acquisition devices may be physical (like a scanner or digital camera) or logical (like an image database). The software elements written to control acquisitions are called Sources and reside primarily in this layer.

The Source transfers data for the application. It uses the format and transfer mechanism agreed upon by the Source and application.

The Source always provides a built-in user interface that controls the device(s) the Source was written to drive. An application can override this and present its own user interface for acquisition, if desired.

Device

This is the location of traditional low-level device drivers. They convert device-specific commands into hardware commands and actions specific to the particular device the driver was written to accompany. Applications that use TWAIN no longer need to ship device drivers because they are part of the Source.

TWAIN is not concerned with the device layer at all. The Source hides the device layer from the application. The Source provides the translation from TWAIN operations and interactions with the Source's user interface into the equivalent commands for the device driver that cause the device to behave as desired.

Note: The Protocol layer is the most thoroughly and rigidly defined to allow precise communications between applications and Sources. The information in this document concentrates on the Protocol and Acquisition layers.

The User Interface to TWAIN

When an application uses TWAIN to acquire data, the acquisition process may be visible to the application's users in the following three areas:

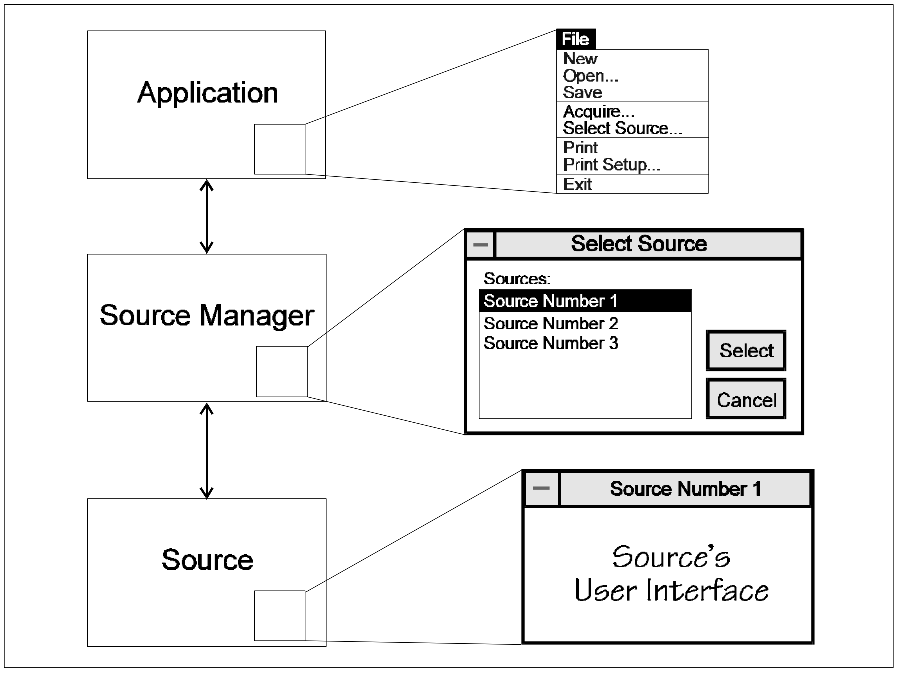


Figure 2-2. Data Acquisition Process

The Application

The user needs to select the device from which they intend to acquire the data. They also need to signal when they are ready to have the data transferred. To allow this, TWAIN strongly recommends the application developer add two options to their File menu:

- **Select Source -** to select the device
- **Acquire -** to begin the transfer process

The Source Manager

When the user chooses the Select Source option, the application requests that the Source Manager display its Select Source dialog box. This lists all available devices and allows the user to highlight and select one device. If desired, the application can write its own version of this user interface.

The Source

Every TWAIN-compliant Source provides a user interface specific to its particular device. When the application user selects the Acquire option, the **Source's User Interface** may be displayed. If desired, the application can write its own version of this interface, too.

Communication Between the Elements of TWAIN

Communication between elements of TWAIN is possible through two entry points. They are called DSM_Entry( ) and DS_Entry( ). DSM means Data Source Manager and DS means Data Source.

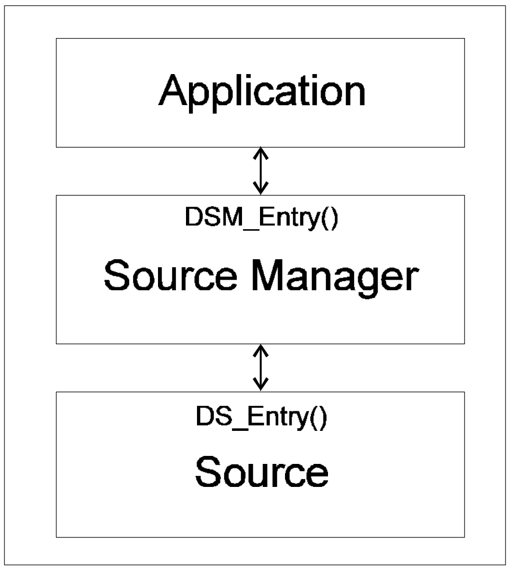


Figure 2-3. Entry Points for Communicating Between Elements

The Application

The goal of the application is to acquire data from a Source. However, applications cannot contact the Source directly. All requests for data, capability information, error information, etc. must be handled through the Source Manager.

Approximately 140 operations are defined by TWAIN. The application sends them to the Source Manager for transmission. The application specifies which element, Source Manager or Source, is the final destination for each requested operation.

The application communicates to the Source Manager through the Source Manager's only entry point, the **DSM_Entry( )** function.

The parameter list of the DSM_Entry function contains:

- An identifier structure providing information about the application that originated the function call
- The destination of this request (Source Manager or Source)
- A triplet that describes the requested operation. The triplet specifies:
  - ✓ Data Group for the Operation (DG_)
  - ✓ Data Argument Type for the Operation (DAT_)
  - ✓ Message for the Operation (MSG_)

  (These are described more in the section called “The Use of Operation Triplets” located later in this chapter.)
- A pointer field to allow the transfer of data

The function call returns a value (the Return Code) indicating the success or failure of the operation.

Written in C code form, the function call looks like this:

On Windows

```
TW_UINT16 FAR PASCAL DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       pTW_IDENTITY   pDest,      // destination of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

On Macintosh

```
FAR PASCAL TW_UINT16 DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       pTW_IDENTITY   pDest,      // destination of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

Note: Data type definitions are covered in Chapter 8 of this document and in the file called TWAIN.H which is shipped on the developer’s disk. (It can also be downloaded from the TWAIN Working Group website.)

The Source Manager

The Source Manager provides the communication path between the application and the Source, supports the user's selection of a Source, and loads the Source for access by the application. Communications from application to Source Manager arrive in the DSM_Entry( ) entry point.

- **If the destination in the DSM_Entry call is the Source Manager -** The Source Manager processes the operation itself.
- **If the destination in the DSM_Entry call is the Source -** The Source Manager translates the parameter list of information, removes the destination parameter and calls the appropriate Source. To reach the Source, the Source Manager calls the Source's **DS_Entry( )** function. TWAIN requires each Source to have this entry point.

Written in C code form, the DS_Entry function call looks like this:

On Windows

```
TW_UINT16 FAR PASCAL DS_Entry
      (pTW_IDENTITY   pOrigin,    // source of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
      );
```

On Macintosh

```
FAR PASCAL TW_UINT16 DS_Entry
      (pTW_IDENTITY   pOrigin,    // source of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
      );
```

In addition, the Source Manager can initiate three operations that were not originated by the application. These operation triplets exist just for Source Manager to Source communications and are executed by the Source Manager while it is displaying its Select Source dialog box. The operations are used to identify the available Sources and to open or close Sources.

The implementation of the Source Manager differs between the supported systems:

On Windows

The Source Manager for Windows is a Dynamic Link Library (DLL).

The Source Manager can manage simultaneous sessions between many applications with many Sources. That is, the same instance of the Source Manager is shared by multiple applications.

On Macintosh

The Source Manager for Macintosh is a code resource.

Each application gets a private copy of the Source Manager and the Source(s) it opens. The separate instances of the Source Manager and Sources can coordinate among themselves.

The Source

The Source receives operations either from the application, via the Source Manager, or directly from the Source Manager. It processes the request and returns the appropriate Return Code (the codes are prefixed with TWRC_) indicating the results of the operation to the Source Manager. If the originator of the operation was the application, then the Return Code is passed back to the application as the return value of its DSM_Entry( ) function call. If the operation was unsuccessful, a Condition Code (the codes are prefixed with TWCC_) containing more specific information is set by the Source. Although the Condition Code is set, it is not automatically passed back. The application must invoke an operation to inquire about the contents of the Condition Code.

The implementation of the Source is the same as the implementation of the Source Manager:

On Windows

The Source is a Dynamic Link Library (DLL) so applications share the same copy of each element.

On Macintosh

The Source is implemented as a Code Resource.

Communication Flowing from Source to Application

The majority of operation requests are initiated by the application and flow to the Source Manager and Source. The Source, via the Source Manager, is able to pass back data and Return Codes.

However, there are four times when the Source needs to interrupt the application and request that an action occur. These situations are:

- **Notify the application that a data transfer is ready to occur.** The time required for a Source to prepare data for a transfer will vary. Rather than have the application wait for the preparation to be complete, the Source just notifies it when everything is ready. The MSG_XFERREADY notice is used for this purpose.
- **Request that the Source's user interface be disabled.** This notification should be sent by the Source to the application when the user clicks on the "Close" button of the Source's user interface. The MSG_CLOSEDSREQ notice is used for this purpose.
- **Notify the application that the OK button has been pressed, accepting the changes the user has made.** This is only used if the Source is opened with DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDSUIONLY. The MSG_CLOSEDSOK notice is used for this purpose.
- **A Device Event has occurred.** This notification is sent by the Source to the Application when a specific event has occurred, but only if the Application gave the Source prior instructions to pass along such events. The MSG_DEVICEEVENT notice is used for this purpose.

These notices are presented to the application in its event (or message) loop. The process used for these notifications is covered more fully in Chapter 3 in the discussion of the application's event loop.

The Use of Operation Triplets

The DSM_Entry( ) and DS_Entry( ) functions are used to communicate operations. An operation is an action that the application or Source Manager invokes. Typically, but not always, it involves using data or modifying data that is indicated by the last parameter (pData) in the function call.

Requests for actions occur in one of these ways:

From	To	Using this function
The application	The Source Manager	DSM_Entry with the pDest parameter set to NULL
The application	The Source (via the Source Manager)	DSM_Entry with the pDest parameter set to point to a valid structure that identifies the Source
The Source Manager	The Source	DS_Entry

The desired action is defined by an operation triplet passed as three parameters in the function call. Each triplet uniquely, and without ambiguity, specifies a particular action. No operation is specified by more than a single triplet. The three parameters that make up the triplet are Data Group, Data Argument Type, and Message ID. Each parameter conveys specific information.

Data Group (DG_xxxx)

Operations are divided into large categories by the Data Group identifier. There are currently only two defined in TWAIN:

- **CONTROL** (The identifier is DG_CONTROL.): These operations involve control of the TWAIN session. An example where DG_CONTROL is used as the Data Group identifier is the operation to open the Source Manager.
- **IMAGE** (The identifier is DG_IMAGE.): These operations work with image data. An example where DG_IMAGE is used as a Data Group is an operation that requests the transfer of image data.
- **AUDIO** (The identifier is DG_AUDIO): These operations work with audio data (supported by some digital cameras). An example where DG_AUDIO is used as a Data Group is an operation that requests the transfer of audio data.

Data Argument Type (DAT_xxxx)

This parameter of the triplet identifies the type of data that is being passed or operated upon. The argument type may reference a data structure or a variable. There are many data argument types. One example is DAT_IDENTITY.

The DAT_IDENTITY type is used to identify a TWAIN element such as a Source. Remember, from the earlier code example, data is typically passed or modified through the pData parameter of the DSM_Entry and DS_Entry. In this case, the pData parameter would point to a data structure of type TW_IDENTITY. Notice that the data argument type begins with DAT_xxxx and the associated data structure begins with TW_xxxx and duplicates the second part of the name. This pattern is followed consistently for most data argument types and their data structures. Any exceptions are noted on the reference pages in Chapters 7 and 8.

Message ID (MSG_xxxx)

This parameter identifies the action that the application or Source Manager wishes to have taken. There are many different messages such as MSG_GET or MSG_SET. They all begin with the prefix of MSG_.

Here are three examples of operation triplets:

The triplet the application sends to the Source Manager to open the Source Manager module is:

DG_CONTROL / DAT_PARENT / MSG_OPENDSM

The triplet that the application sends to instruct the Source Manager to display its Select Source dialog box and thus allow the user to select which Source they plan to obtain data from is:

DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

The triplet the application sends to transfer data from the Source into a file is:

DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET

The State-Based Protocol

The application, Source Manager, and Source must communicate to manage the acquisition of data. It is logical that this process must occur in a particular sequence. For example, the application cannot successfully request the transfer of data from a Source before the Source Manager is loaded and prepared to communicate the request.

To ensure the sequence is executed correctly, the TWAIN protocol defines seven states that exist in TWAIN sessions. A session is the period while an application is connected to a particular Source via the Source Manager. The period while the application is connected to the Source Manager is another unique session. At a given point in a session, the TWAIN elements of Source Manager and Source each occupy a particular state. Transitions to a new state are caused by operations requested by the application or Source. Transitions can be in the forward or backward direction. Most transitions are single-state transitions. For example, an operation moves the Source Manager from State 1 to State 2 not from State 1 to State 3. (There are situations where a two-state transition may occur. They are discussed in Chapter 3.)

When viewing the state-based protocol, it is helpful to remember:

States 1, 2, and 3

- Are occupied only by the Source Manager.
- The Source Manager never occupies a state greater than State 3.

States 4, 5, 6, and 7

- Are occupied exclusively by Sources.
- A Source never has a state less than 4 if it is open. If it is closed, it has no state.
- If an application uses multiple Sources, each connection is a separate session and each open Source “resides” in its own state without regard for what state the other Sources are in.

The State Transition Diagram looks like this:

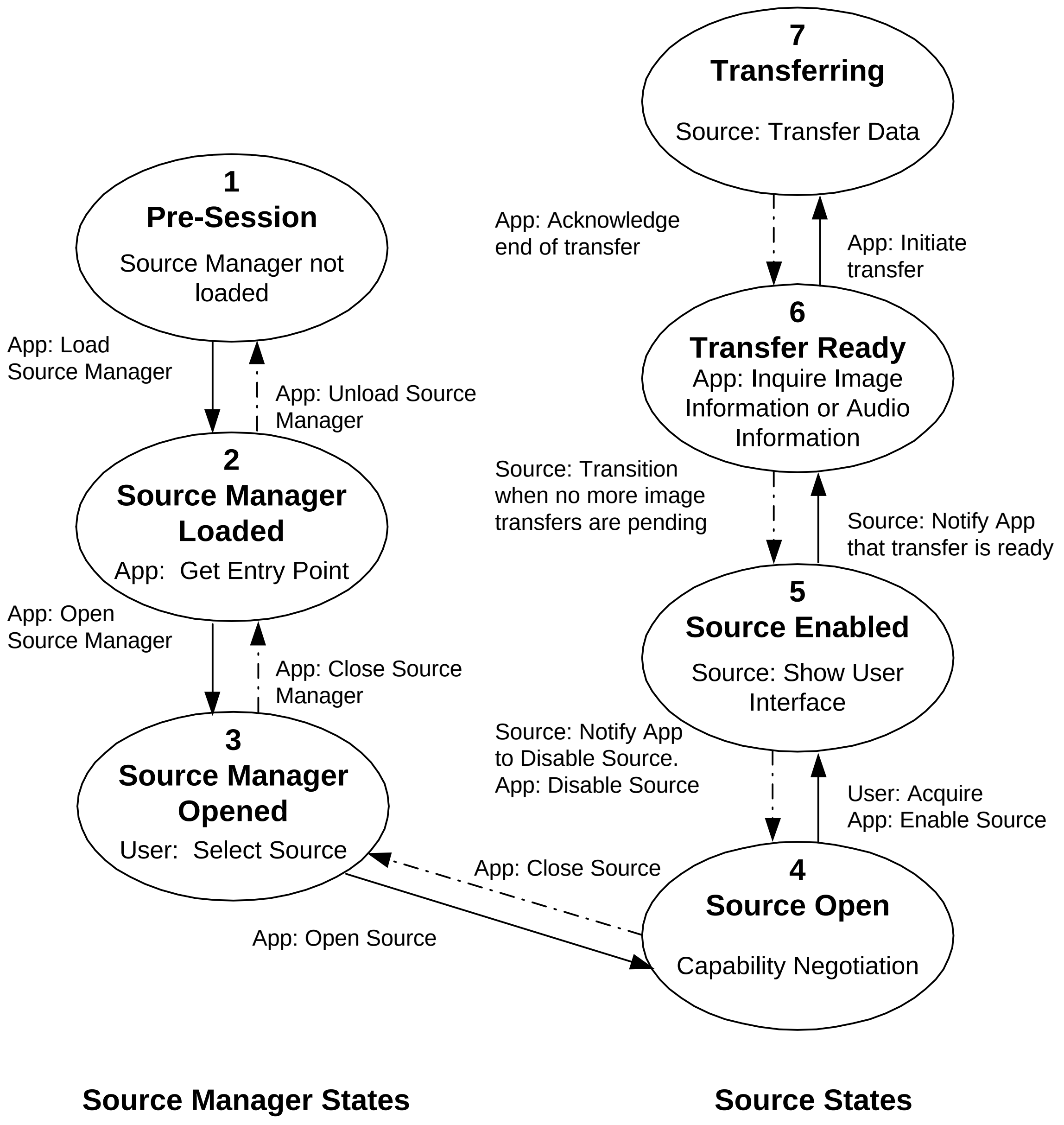

7
Transferring
Source: Transfer Data
1
Pre-Session
Source Manager not loaded
App: Acknowledge end of transfer
App: Initiate transfer
6
Transfer Ready
App: Inquire Image Information or Audio Information
App: Load Source Manager
App: Unload Source Manager
2
Source Manager Loaded
App: Get Entry Point
Source: Transition when no more image transfers are pending
Source: Notify App that transfer is ready
5
Source Enabled
Source: Show User Interface
App: Open Source Manager
App: Close Source Manager
3
Source Manager Opened
User: Select Source
Source: Notify App to Disable Source. App: Disable Source
User: Acquire App: Enable Source
App: Close Source
4
Source Open
Capability Negotiation
App: Open Source
Source Manager States
Source States

Figure 2-4. State Transition Diagram

The Description of the States

The following sections describe the states.

State 1 - Pre-Session

The Source Manager “resides” in State 1 before the application establishes a session with it.

At this point, the Source Manager code has been installed on the disk but typically is not loaded into memory yet.

The only case where the Source Manager could already be loaded and running is under Windows because the implementation is a DLL (hence, the same instance of the Source Manager can be shared by multiple applications). If that situation exists, the Source Manager will be in State 2 or 3 with the application that loaded it.

State 2 - Source Manager Loaded

The Source Manager now is loaded into memory. It is not open yet.

At this time, the Source Manager is prepared to accept other operation triplets from the application.

State 3 - Source Manager Open

The Source Manager is open and ready to manage Sources.

The Source Manager is now prepared to provide lists of Sources, to open Sources, and to close Sources.

The Source Manager will remain in State 3 for the remainder of the session until it is closed. The Source Manager refuses to be closed while the application has any Sources open.

State 4 - Source Open

The Source has been loaded and opened by the Source Manager in response to an operation from the application. It is ready to receive operations.

The Source should have verified that sufficient resources (i.e. memory, device is available, etc.) exist for it to run.

The application can inquire about the Source’s capabilities (i.e. levels of resolution, support of color or black and white images, automatic document feeder available, etc.). The application can also set those capabilities to its desired settings. For example, it may restrict a Source capable of providing color images to transferring black and white only.

Note: Inquiry about a capability can occur while the Source is in States 4, 5, 6, or 7. But, an application can set a capability only in State 4 unless special permission is negotiated between the application and Source.

State 5 - Source Enabled

The Source has been enabled by an operation from the application via the Source Manager and is ready for user-enabled transfers.

If the application has allowed the Source to display its user interface, the Source will do that when it enters State 5.

State 6 - Transfer is Ready

The Source is ready to transfer one or more data items (images) to the application.

The transition from State 5 to 6 is triggered by the Source notifying the application that the transfer is ready.

Before initiating the transfer, the application must inquire information about the image (resolution, image size, etc.). If the Source supports audio, then before transferring the image, the Application must transfer all the audio snippets that are associated with the image.

It is possible for more than one image to be transferred in succession. This topic is covered thoroughly in Chapter 4.

State 7 - Transferring

The Source is transferring the image to the application.

The transfer mechanism being used was negotiated during State 4.

The transfer will either complete successfully or terminate prematurely. The Source sends the appropriate Return Code indicating the outcome.

Once the Source indicates that the transfer is complete, the application must acknowledge the end of the transfer.

Capabilities

One of TWAIN's benefits is it allows applications to easily interact with a variety of acquisition devices. Devices can provide image or audio data. For instance,

- Some devices have automatic document feeders.
- Some devices are not limited to one image but can transfer multiple images.
- Some devices support color images.
- Some devices offer a variety of halftone patterns.
- Some devices support a range of resolutions while others may offer different choices.
- Some devices allow the recording of audio data associated with an image.

Developers of applications need to be aware of a Source's capabilities and may influence the capabilities that the Source offers to the application's users. To do this, the application can perform **capability negotiation**. The application generally follows this process:

1. **Determine** if the selected Source supports a particular capability.
2. **Inquire** about the Current Value for this capability. Also, inquire about the capability's Default Value and the set of Available Values that are supported by the Source for that capability.
3. **Request** that the Source set the Current Value to the application's desired value. The Current Value will be displayed as the current selection in the Source's user interface.
4. **Limit**, if needed, the Source's Available Values to a subset of what would normally be offered. For instance, if the application wants only black and white data, it can restrict the Source to transmit only that. If a limitation effects the Source's user interface, the Source should modify the interface to reflect those changes. For example, it may gray out options that are not available because of the application's restrictions.

5. **Verify** that the new values have been accepted by the Source.

TWAIN capabilities are divided into three groups:

- **CAP_xxxx:** Capabilities whose names begin with CAP are capabilities that could apply to any general Source. Such capabilities include use of automatic document feeders, identification of the creator of the data, etc.
- **ICAP_xxxx:** Capabilities whose names begin with ICAP are capabilities that apply to image devices. The "I" stands for image. (When TWAIN is expanded to support other data transfer such as text or fax data, there will be TCAPs and FCAPs in a similar style.)
- **ACAP_xxxx:** Capabilities whose names begin with ACAP are capabilities that apply to devices that support audio. The "A" stands for audio.

Capability Containers

Capabilities exist in many varieties but all have a Default Value, Current Value, and may have other values available that can be supported if selected. To help categorize the supported values into clear structures, TWAIN defines four types of **containers** for capabilities.

Name of the Data Structure for the Container	**Type of Contents**
TW_ONEVALUE	A single value whose current and default values are coincident. The range of available values for this type of capability is simply this single value. For example, a capability that indicates the presence of a document feeder could be of this type.
TW_ARRAY	A rectangular array of values that describe a logical item. It is similar to the TW_ONEVALUE because the current and default values are the same and there are no other values to select from. For example, a list of the names, such as the supported capabilities list returned by the CAP_SUPPORTEDCAPS capability, would use this type of container.
TW_RANGE	Many capabilities allow users to select their current value from a range of regularly spaced values. The capability can specify the minimum and maximum acceptable values and the incremental step size between values. For example, resolution might be supported from 100 to 600 in steps of 50 (100, 150, 200, ..., 550, 600).
TW_ENUMERATION	This is the most general type because it defines a list of values from which the Current Value can be chosen. The values do not progress uniformly through a range and there is not a consistent step size between the values. For example, if a Source's resolution options did not occur in even step sizes then an enumeration would be used (for example, 150, 400, and 600).

In general, most capabilities can have more than one of these containers applied to them depending on how the particular Source implements the capability. The data structure for each of these containers is defined in Chapter 8. A complete table with all defined capabilities is located in Chapter 9. A few of the capabilities must be supported by the application and Source. The remainder of the capabilities are optional.

Capability Negotiation and Container Types

It is very important for Application and Data Source developers to note that Container types are dictated by the Data Source in all cases where a value is queried. Also the allowable container types of each capability are clearly defined in Chapter 9 of the TWAIN Specification. The only time it is appropriate for the calling Application to specify a container type is during the **MSG_SET** operation. At that time, the Application must also consider the allowable containers and types for the particular capability.

Capability Containers and String Values

The only containers that can possibly hold a string are the following:

TW_ENUMERATION
TW_ARRAY
TW_ONEVALUE

It is not possible or useful to use this type in a **TW_RANGE.** in fact there is no case where a capability has been defined in Chapter 9 of the TWAIN Specification where a **TW_RANGE** is allowed for a **TW_STRxxxx** type of value.

There are four types of TWAIN strings defined for developer use:

TW_STR32
TW_STR64
TW_STR128
TW_STR256

As of version 1.7, only the following capabilities accept strings:

CAP_AUTHOR, TW_ONEVALUE, TW_STR128
CAP_CAPTION, TW_ONEVALUE, TW_STR255
CAP_TIMEDATE, TW_ONEVALUE, TW_STR32
ICAP_HALFTONES, TW_ONEVALUE / TW_ENUMERATION / TW_ARRAY, TW_STR32

The definition of the various container types could be confusing. For example, the definition of a TW_ONEVALUE is as follows:

```
/* TWON_ONEVALUE. Container for one value. */
typedef struct {
   TW_UINT16  ItemType;
   TW_UINT32  Item;
} TW_ONEVALUE, FAR * pTW_ONEVALUE;
```

At first glance, it is tempting to try placing the string into this container by assigning “Item” to be a pointer. This is not at all consistent with the implementation of other structures in the specification and introduces a host of problems concerning management of the memory occupied by the string. (See TW_IDENTITY for consistent TWAIN string use)

The correct and consistent method of holding a string in a TWAIN container is to ensure the string is embedded in the container itself. Either a new structure is defined within the developers code, or the added size is considered when allocating the container.

The following examples are designed to demonstrate possible methods of using TWAIN Strings in Containers. These examples are suitable for demonstration only, and require refinement to be put to real use.

Example 1:
TW_ONEVALUE structure defined for holding a TW_STR32 value

```
/* TWON_ONEVALUESTR32. Container for one value holding TW_STR32. */
typedef struct {
   TW_UINT16  ItemType;
   TW_STR32  Item;
} TW_ONEVALUESTR32, FAR * pTW_ONEVALUESTR32;
```

Note: Pay attention to two-byte structure packing when defining custom container structures.

This clearly demonstrates where the memory is allocated and where the string resides. The data source does not have to be concerned with how the string is managed locally, and the application does not have to be concerned with managing the string memory or contents.

Example 2:
TW_ONEVALUE structure allocated and filled with consideration of holding a TW_STR32 value (Windows Example)

```
HGLOBAL AllocateAndFillOneValueStr32( const pTW_STR32 pInString )
{
   DWORD dwContainerSize = 0l;
   HGLOBAL hContainer = NULL;
   pTW_ONEVALUE pOneValue = NULL;
   pTW_STR32 pString = NULL;

   assert(pInString);

   // Note: This calculation will yield a size approximately one
   // pointer larger than that required for this container
   // (sizeof(TW_UINT32)).  For simplicity the size difference
   // is negligible.  The first TW_STR32 item shall be located
   // immediately after the pEnum->DefaultIndex member.
      dwContainerSize = sizeof(TW_ONEVALUE) + sizeof(TW_STR32);
      hContainer = GlobalAlloc( GPTR, dwContainerSize );
      if(hContainer)
   {
      pOneValue = (pTW_ONEVALUE)GlobalLock(hContainer);
      if(pOneValue)
      {
         pOneValue->ItemType = TWTY_STR32;
         pString = (pTW_STR32)&pOneValue->Item;

         memcpy(pString, pInString, sizeof(TW_STR32));
```

```
            GlobalUnlock(hContainer);
            pOneValue = NULL;
            pString = NULL;
         }
      }
      return hContainer;
   }
```

Example 3:
TW_ENUMERATION structure allocated with consideration of holding TW_STR32 values (Windows Example)

```
   HGLOBAL AllocateEnumerationStr32( TW_UINT32 unNumItems )
   {
      DWORD dwContainerSize = 0l;
      HGLOBAL hContainer = NULL;
      pTW_ENUMERATION pEnum = NULL;

      // Note: This calculation will yield a size approximately
      // one pointer larger than that required for this container
      // (sizeof(pTW_UINT8)).  For simplicity the size difference is
      // negligible.  The first TW_STR32 item shall be located
      // immediately after the pEnum->DefaultIndex member.

   dwContainerSize = sizeof(TW_ENUMERATION) + ( sizeof(TW_STR32) *
   unNumItems);

   hContainer = GlobalAlloc( GPTR, dwContainerSize );
      if(hContainer)
      {
         pEnum = (pTW_ENUMERATION) GlobalLock(hContainer);
         if(pEnum)
         {
            pEnum->ItemType = TWTY_STR32;
            pEnum->NumItems = unNumItems;

   GlobalUnlock(hContainer);
            pEnum = NULL;
         }
      }
      return hContainer;
   }
```

Example 4: Indexing a string from an Enumeration Container

```
   pTW_STR128 IndexStr128FromEnumeration( pTW_ENUMERATION pEnum, TW_UINT32
   unIndex)
   {
      BYTE *pBegin = (BYTE *)&pEnum->ItemList[0];
      assert(pEnum->NumItems > unIndex);
      assert(pEnum->ItemType == TWTY_STR128);

      pBegin += (unIndex * sizeof(TW_STR128));
      return (pTW_STR128)pBegin;
   }
```

Modes Available for Data Transfer

There are three different modes that can be used to transfer data from the Source to the application: native, disk file, and buffered memory. (At this time, TWAIN support for audio only allows native and disk file transfers.)

Native

Every Source must support this transfer mode. It is the default mode and is the easiest for an application to implement. However, it is restrictive (i.e. limited to the DIB or PICT formats and limited by available memory).

The format of the data is platform-specific:

- Windows: DIB (Device-Independent Bitmap)
- Macintosh: A handle to a Picture

The Source allocates a single block of memory and writes the image data into the block. It passes a pointer to the application indicating the memory location. The application is responsible for freeing the memory after the transfer.

Disk File

A Source is not required to support this transfer mode but it is recommended.

The application creates the file to be used in the transfer and ensures that it is accessible by the Source for reading and writing.

A capability exists that allows the application to determine which file formats the Source supports. The application can then specify the file format and file name to be used in the transfer.

The disk file mode is ideal when transferring large images that might encounter memory limitations with Native mode. Disk File mode is simpler to implement than the buffered mode discussed next. However, Disk File mode is a bit slower than Buffered Memory mode and the application must be able to manage the file after creation.

Buffered Memory

Every Source must support this transfer mode.

The transfer occurs through memory using one or more buffers. Memory for the buffers are allocated and deallocated by the application.

The data is transferred as an unformatted bitmap. The application must use information available during the transfer (TW_IMAGEINFO and TW_IMAGEMEMXFER) to learn about each individual buffer and be able to correctly interpret the bitmap.

If using the Native or Disk File transfer modes, the transfer is completed in one action. With the Buffered Memory mode, the application may need to loop repeatedly to obtain more than one buffer of data.

Buffered Memory transfer offers the greatest flexibility, both in data capture and control. However, it is the least simple to implement.

3

Application Implementation

This chapter provides the basic information needed to implement TWAIN at a minimum level. In this chapter, you will find information on:

Chapter Contents

Advanced topics are discussed in Chapter 4. They include how to take advantage of Sources that offer automatic feeding of multiple images.

Levels of TWAIN Implementation

Application developers can choose to implement TWAIN features in their application along a range of levels.

- **At the minimum level:** The application does not have to take advantage of capability negotiation or transfer mode selection. Using TWAIN defaults, it can just acquire a single image in the Native mode.
- **At a greater level:** The application can negotiate with the Source for desired capabilities or image characteristics and specify the transfer arrangement. This gives the application more control over the type of image it receives. To do this, developers should follow the instructions provided in this chapter and use information from Chapter 4, as well.
- **At the highest level:** An application may choose to negotiate capabilities, select transfer mode, and create / present its own user interfaces instead of using the built-in ones provided with the Source Manager and Source. Again, refer to this chapter and Chapter 4.

Installation of the Source Manager Software

For a TWAIN-compliant application or Source to work properly, a Source Manager **must** be installed on the host system. To guarantee that a Source Manager is available, ship a copy of the latest Source Manager on your product's distribution disk and provide the user with an installer or installation instructions as suggested below. To ensure that the most recent version of the Source Manager is available to you and your user on their computer, you must do the following:

1. Look for a Source Manager:
   a. **On Windows systems**: Look for the file names TWAIN.DLL, TWAIN_32.DLL, TWUNK_16.EXE, and TWUNK_32.EXE in the Windows directory (this is typically C:\Windows on Windows 3.1/95/98, and C:\Winnt on Windows NT).
   b. **On Macintosh systems**: Look for the file name "Source Manager" in the TWAIN folder. On System 6, the TWAIN folder lives in the System Folder. On System 7, the TWAIN folder lives in the Preferences folder which lives in the System Folder. (Note, the term "Source Manager" may be localized for other languages.)
2. If no Source Manager is currently installed, install the Source Manager sent out with your application.
3. If a Source Manager already exists, check the version of the installed Source Manager. If the version provided with your application is more recent, rename the existing one as follows and install the Source Manager you shipped. To rename the existing Source Manager:
   a. **On Windows systems**: Rename the four files to be TWAIN.BAK, TWAIN_32.BAK, TWUNK_16.BAK, and TWUNK_32.BAK.
   b. **On Macintosh systems**: Rename it to Source Manager Old.

How to Install the Source Manager on Microsoft Windows Systems

To allow the comparison of Source Manager versions, the Microsoft Windows Source Manager DLL has version information built into it which conforms to the Microsoft File Version Stamping specification. Application developers are strongly encouraged to take advantage of this in their installation programs. Microsoft provides the File Version Stamping Library, VER.DLL, which should be used to install the Source Manager.

VER.DLL, VER.LIB and VER.H are included in this Toolkit; VER.DLL may be freely copied and distributed with your installation program. Of course, your installation program will have to link to this DLL to use it. Documentation on the File Version Stamping Library API can be found in the Microsoft Windows 3.1 SDK. VER.DLL can be used under Windows 3.0 as well as 3.1.

The following code fragment demonstrates how the VerInstallFile( ) function provided in VER.DLL can be used to install the Source Manager into the user's Windows directory.

Note that the following example assumes that your installation floppy disk is in the A: drive and the Source Manager is in the root of the installation disk.

```
#include "windows.h"
#include "ver.h"
#include "stdio.h"

// Max file name length is based on 8 dot 3 file name convention.
#define MAXFNAMELEN 12
// Max path name length is based on GetWindowsDirectory()
// documentation.
#define MAXPATHLEN 144

VOID InstallWinSM ( VOID )
{
   DWORD  dwInstallResult;
   WORD   wTmpFileLen = MAXPATHLEN;
   WORD   wLen;

   char   szSrcDir[MAXPATHLEN];
   char   szDstDir[MAXPATHLEN];
   char   szCurDir[MAXPATHLEN];
   char   szTmpFile[MAXPATHLEN];

   wLen = GetWindowsDirectory( szDstDir, MAXPATHLEN );
   if (!wLen || wLen>MAXPATHLEN)
   {
      return;   // failure getting Windows dir
   }

   strcpy( szCurDir, szDstDir );
   strcpy( szSrcDir, "a:\\" );

   dwInstallResult = VerInstallFile( VIFF_DONTDELETEOLD,
                                     "TWAIN_32.DLL",
                                     "TWAIN_32.DLL",
                                     szSrcDir,
                                     szDstDir,
                                     szCurDir,
                                     szTmpFile,
                                     &wTmpFileLen );

   // If VerInstallFile() left a temporary copy of the new
   // file in DstDir be sure to delete it. This happens
   // when a more recent version is already installed.
   if ( dwInstallResult & VIF_TEMPFILE  &&
        ((wTmpFileLen - MAXPATHLEN) > MAXFNAMELEN) )
   {
      // when dst path is root it already ends in ‘\’
      if (szDstDir[wLen-1] != '\\')
      {
         strcat( szDstDir, "\\" );
      }
      strcat( szDstDir, szTmpFile );
      remove( szDstDir );
   }
}
```

You should enhance the above code so that it handles the other three files (TWAIN.DLL, TWUNK_16.EXE, and TWUNK_32.EXE), as well as fixing it to handle low memory and other error conditions, as indicated by the dwInstallResult return code. Also note that the above code does not leave a backup copy of the user's prior Source Manager on their disk, but you should do this. Copy the older versions to TWAIN.BAK, TWAIN_32.BAK, TWUNK_16.BAK, and TWUNK_32.BAK.

How to Install the Source Manager on Macintosh Systems

To perform the comparison of Source Manager versions, use the vers resource. Use the following algorithm to develop code for installing the Source Manager on the Macintosh:

```
if (there is not a Source Manager installed) or
   (if installed Source Manager is older)
   if (the current system predates 7.0)
      if (there isn't a "TWAIN" folder in the System Folder)
         Create a "TWAIN" folder in the System Folder
      copy the Source Manager into the "TWAIN" folder in System Folder
   else
      if (there isn't a "TWAIN" folder in the Preferences Folder)
         Create a "TWAIN" folder in the Preferences folder
      copy the Source Manager into the "TWAIN" folder
```

Changes Needed to Prepare for a TWAIN Session

There are three areas of the application that must be changed before a TWAIN session can even begin. The application developer must:

1. Alter the application's user interface to add Select Source and Acquire menu choices
2. Include the file called TWAIN.H in your application
3. Alter the application's event loop

Alter the Application's User Interface to Add Select Source and Acquire Options

As mentioned in the Technical Overview chapter, the application should include two menu items in its File menu: **Select Source...** and **Acquire....** It is strongly recommended that you use these phrases since this consistency will benefit all users.

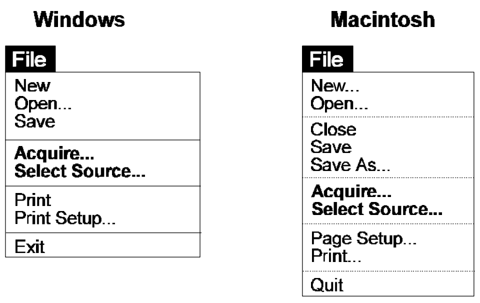


Figure 3-1. User Interface for Selecting a Source and Acquiring Options

Note the following:

When this is selected:	**The application does this:**
Select Source...	The application requests that the Source Manager's Select Source Dialog Box appear (or it may display its own version). After the user selects the Source they want to use, control returns to the application.
Acquire...	The application requests that the Source display its user interface. (Again, the application can create its own version of a user interface or display no user interface.)

Detailed information on the operations used by the application to successfully acquire data is provided later in this chapter in the section called "Controlling a TWAIN Session from your Application."

Include the TWAIN.H File in Your Application

The TWAIN.H file that is shipped with this TWAIN Developer's Toolkit contains all of the critical definitions needed for writing a TWAIN-compliant application or Source. Be sure to include it in your application's code and print out a copy to refer to while reading this chapter.

The TWAIN.H file contains:

Category	**Prefix for each item**
Data Groups	DG_
Data Argument Types	DAT_
Messages	MSG_
Capabilities	CAP_, ICAP_, or ACAP_
Return Codes	TWRC_
Condition Codes	TWCC_
Type Definitions	TW_
Structure Definitions	TW_
Entry points	These are DSM_Entry and DS_Entry

In addition, there are many constants defined in TWAIN.H which are not listed here.

Alter the Application's Event Loop

Events include activities such as key clicks, mouse events, periodic events, accelerators, etc. Every TWAIN-compliant application, whether on Macintosh or Windows, needs an event loop. (On Windows, these actions are called messages but that can be confusing because TWAIN uses the term messages to describe the third parameter of an operation triplet. Therefore, we will refer to these key clicks, etc. as events in this section generically for both Windows and Macintosh.)

During a TWAIN session, the application opens one or more Sources. However, even if several Sources are open, the application should only have one Source enabled at any given time. That is the Source from which the user is attempting to acquire data.

Altering the event loop serves three purposes:

- Passing events from the application to the Source so it can respond to them
- Notifying the application when the Source is ready to transfer data or have its user interface disabled
- Notifying the application when a device event occurs.

Event Loop Modification - Events in State 4

Please note that with TWAIN 1.8 and the addition of the DG_CONTROL / DAT_NULL / MSG_DEVICEEVENT message, it is possible to receive events after the Source has been opened but before it has been enabled (State 4). However, these events will not be sent from the Source to the Application unless the Application has negotiated for specific events using CAP_DEVICEEVENTS. Events posted in this way must use the hWnd passed to them by the DG_CONTROL / DAT_PARENT / MSG_OPENDS message. Sources are required to have all device events turned off when they are opened to support backward compatiblity with older TWAIN applications.

Event Loop Modification - Passing events (The first purpose)

While a Source is enabled, all events are sent to the application's event loop. Some of the events may belong to the application but others belong to the enabled Source. To ensure that the Source receives and processes its events, the following changes are required:

The application **must** send all events that it receives in its event loop to the Source as long as the Source is enabled. The application uses:

DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT

The TW_EVENT data structure used looks like this:

```
typedef struct {
   TW_MEMREF     pEvent;     /* Windows pMSG or MAC pEvent   */
   TW_UINT16     TWMessage;  /* TW message from Source to    */
                             /* the application              */
} TW_EVENT, FAR *pTW_EVENT;
```

The pEvent field points to the EventRecord (Macintosh) or message structure (Windows).

The Source receives the event from the Source Manager and determines if the event belongs to it.

- **If it does**, the Source processes the event. It then sets the Return Code to TWRC_DSEVENT to indicate it was a Source event. In addition, it should set the TWMessage field of the TW_EVENT structure to MSG_NULL.
- **If it does not**, the Source sets the Return Code to TWRC_NOTDSEVENT meaning it is not a Source event. In addition, it should set the TWMessage field of the TW_EVENT structure to MSG_NULL. The application receives this information from DSM_Entry and should process the event in its event loop as normal.

On Macintosh only, the application must periodically send NULL events to the Source to allow notifications from Source to application.

Event Loop Modification - Notifications from Source to application (The second and third purpose)

When the Source has data ready for a data transfer or it wishes to request that its user interface be disabled, it needs to communicate this information to the application asynchronously.

These notifications appear in the application's event loop. They are contained in the TW_EVENT.TWMessage field. The four notices of interest are:

- MSG_XFERREADY to indicate data is ready for transfer
- MSG_CLOSEDSREQ to request that the Source's user interface be disabled
- MSG_CLOSEDSOK to request that the Source's user interface be disabled (special case for use with DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDSUIONLY).
- MSG_DEVICEEVENT to report that a device event has occurred.

Therefore, the application's event loop must always check the TW_EVENT.TWMessage field following a DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT call to determine if it is the simple MSG_NULL or critical MSG_XFERREADY or MSG_CLOSEDSREQ. Information about how the application should respond to these two special notices is detailed later in this chapter in the "Controlling a TWAIN Session from your Application" section.

How to Modify the Event Loop for Microsoft Windows

This section illustrates typical modifications needed in an Microsoft Windows application to support TWAIN-connected Sources.

```
TW_EVENT   twEvent;
TW_INT16   rc;

while (GetMessage ( (LPMSG) &msg, NULL, 0, 0) )
{
      if Source is enabled
      {
            twEvent.pEvent = (TW_MEMREF)&msg;
            twEvent.TWMessage = MSG_NULL;
            rc = (*pDSM_Entry) (pAppId,
                                pSourceId,
                                DG_CONTROL,
                                DAT_EVENT,
                                MSG_PROCESSEVENT,
                               (TW_MEMREF)&twEvent);

            // check for message from Source
            switch (twEvent.TWMessage)
            {
                  case MSG_XFERREADY:
                        SetupAndTransferImage(NULL);
                        break;

                  case MSG_CLOSEDSREQ:
                        DisableAndCloseSource(NULL);
                        break;
```

```
                    case MSG_NULL:
                          // no message was returned from the source
                          break;
              }
              if (rc == TWRC_NOTDSEVENT)
              {
                    TranslateMessage( (LPMSG) &msg);
                    DispatchMessage( (LPMSG) &msg);
              }
        }
  }
```

Note: Source writers are advised to keep stack space usage to a minimum. Application writers should be also be aware that, in the Windows environment, sources run in their calling application's data space. They depend upon the application to reserve enough stack space for the source to be able to perform its various functions. For this reason, applications should define enough stack space in their linker DEF files for the sources that they might use.

How to Modify the Event Loop for Macintosh

This section illustrates typical modifications needed in a Macintosh application to support TWAIN-connected Sources.

```
TW_EVENT      twEvent;
TW_INT16      rc;
EventRecord   theEvent;
while (!Done){
      If Source is Enabled{
             //Send periodic NULL events to the Source
             twEvent.pEvent = NULL;
             twEvent.TWMessage = MSG_NULL;
             rc = (*pDSM_Entry) (pAppID,
                          pSourceID,
                          DG_CONTROL,
                          DAT_EVENT,
                          MSG_PROCESSEVENT,
                          (TW_MEMREF)&twEvent);
             //check for message from Source
             switch (twEvent.TWMessage){
                   case MSG_XFERREADY:
                          SetupImage(NULL);
                          break;
                   case MSG_CLOSEDSREQ:
                          DisableSource(NULL);
                          break;
                   case MSG_NULL:
                          //no message was returned from the Source
                          break;
             }
      }
      if (GetNextEvent(everyEvent, &theEvent) ){   //or WaitNextEvent()
             If Source is Enabled{
                   twEvent.pEvent = &theEvent;
                   twEvent.TWMessage = MSG_NULL;
                   rc = (*pDSM_Entry) (pAppID,
                          pSourceID,
                          DG_CONTROL,
                          DAT_EVENT,
                          MSG_PROCESSEVENT,
                         (TW_MEMREF)&twEvent);
                   //check for message from Source
                   switch (twEvent.TWMessage){
                         case MSG_XFERREADY:
                               SetupImage(NULL);
                               break;
                         case MSG_CLOSEDSREQ:
                               DisableSource(NULL);
                               break;
                         case MSG_NULL:
                               //no message was returned from the Source
                               break;
                   }
                   if (rc == TWRC_NOTDSEVENT)
                         Message=DealWithEvent(&theEvent);
             }
      } else
             Message=DealWithEvent(&theEvent);
}
```

The DSM_Entry Call and Available Operation Triplets

As described in the Technical Overview chapter, all actions that the application invokes on the Source Manager or Source are routed through the Source Manager. The application passes the request for the action to the Source Manager via the DSM_Entry function call which contains an operation triplet describing the requested action. In code form, the DSM_Entry function looks like this:

On Windows:

```
TW_UINT16 FAR PASCAL DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       pTW_IDENTITY   pDest,      // destination of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

On Macintosh:

```
FAR PASCAL TW_UINT16 DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       pTW_IDENTITY   pDest,      // destination of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

The DG, DAT, and MSG parameters contain the operation triplet. The parameters must follow these rules:

pOrigin

References the application's TW_IDENTITY structure. The contents of this structure must not be changed by the application from the time the connection is made with the Source Manager until it is closed.

pDest

Set to NULL if the operation's final destination is the Source Manager.

Otherwise, set to point to a valid TW_IDENTITY structure for an open Source.

DG_xxxx

Data Group of the operation. Currently, only DG_CONTROL, DG_IMAGE, and DG_AUDIO are defined. Custom Data Groups can be defined.

DAT_xxxx

Designator that uniquely identifies the type of data "object" (structure or variable) referenced by pData.

MSG_xxxx

Message specifies the action to be taken.

pData

Refers to the TW_xxxx structure or variable that will be used during the operation. Its type is specified by the DAT_xxxx. This parameter should always be typecast to TW_MEMREF when it is being referenced.

Operation Triplets - Application to Source Manager

There are nine operation triplets that can be sent from the application to be consumed by the Source Manager. They all use the DG_CONTROL data group and they use three different data argument types: DAT_IDENTITY, DAT_PARENT, and DAT_STATUS. The following table lists the data group, data argument type, and messages that make up each operation. The list is in alphabetical order not the order in which they are typically called by an application. Details about each operation are available in reference format in Chapter 7.

Control Operations from Application to Source Manager

DG_CONTROL / DAT_IDENTITY

MSG_CLOSEDS :	Prepare specified Source for unloading
MSG_GETDEFAULT :	Get identity information of the default Source
MSG_GETFIRST :	Get identity information of the first available Source
MSG_GETNEXT :	Get identity of the next available Source
MSG_OPENDS :	Load and initialize the specified Source
MSG_USERSELECT :	Present “Select Source” dialog

DG_CONTROL / DAT_PARENT

MSG_CLOSEDSM :	Prepare Source Manager for unloading
MSG_OPENDSM :	Initialize the Source Manager

DG_CONTROL / DAT_STATUS

MSG_GET :	Return Source Manager’s current Condition Code

Operation Triplets - Application to Source

The next group of operations are sent to a specific Source by the application. These operations are still passed via the Source Manager using the DSM_Entry call. The first set of triplets use the DG_CONTROL identification for their data group. These are operations that could be performed on any kind of TWAIN device. The second set of triplets use the DG_IMAGE identification for their data group which indicates these operations are specific to image data. Details about each operation are available in reference format in Chapter 7.

Control Operations from Application to Source

DG_CONTROL / DAT_CAPABILITY

MSG_GET :	Return a Capability’s valid value(s) including current and default values
MSG_GETCURRENT :	Get a Capability’s current value
MSG_GETDEFAULT :	Get a Capability’s preferred default value (Source specific)
MSG_RESET :	Change a Capability’s current value to its TWAIN-defined default (See Chapter 9)
MSG_SET :	Change a Capability’s current and / or available value(s)

DG_CONTROL / DAT_DEVICEEVENT

MSG_GET :	Get an event from the Source (issue this call only in response to a DG_CONTROL / DAT_NULL / MSG_DEVICEEVENT from the Source)

DG_CONTROL / DAT_EVENT

MSG_PROCESSEVENT :	Pass an event to the Source from the application

DG_CONTROL / DAT_FILESYSTEM

MSG_AUTOMATICCAPTUREDIRECTORY :	Select directory to receive automatically captured images
MSG_CHANGEDIRECTORY :	Change the current domain, host, directory, or device.
MSG_COPY :	Copy files
MSG_CREATEDIRECTORY :	Create a directory
MSG_DELETE :	Delete a file or directory
MSG_FORMATMEDIA :	Format a storage device
MSG_GETCLOSE :	Close a context created by a call to MSG_GETFILEFIRST
MSG_GETFIRSTFILE :	Get the first file in a directory
MSG_GETINFO :	Get information about the current file context
MSG_RENAME :	Rename a file

DG_CONTROL / DAT_PASSTHRU / MSG_PASSTHRU

MSG_PASSTHRU :	Special command for the use by Source vendors when writing diagnostic Applications

DG_CONTROL / DAT_PENDINGXFERS

MSG_ENDXFER :	Application acknowledges or requests the end of data transfer
MSG_GET :	Return the number of transfers the Source is ready to supply
MSG_RESET :	Set the number of pending transfers to zero

DG_CONTROL / DAT_SETUPFILEXFER

MSG_GET :	Return info about the file that the Source will write the acquired data into
MSG_GETDEFAULT :	Return the default file transfer information
MSG_RESET :	Reset current file information to default values
MSG_SET :	Set file transfer information for next file transfer

DG_CONTROL / DAT_SETUPMEMXFER

MSG_GET :	Return Source’s preferred, minimum, and maximum transfer buffer sizes

DG_CONTROL / DAT_STATUS

MSG_GET :	Return the current Condition Code from specified Source

DG_CONTROL / DAT_USERINTERFACE

MSG_DISABLEDS :	Cause Source’s user interface to be taken down
MSG_ENABLEDS :	Cause Source to prepare to display its user interface

DG_CONTROL / DAT_XFERGROUP

MSG_GET :	Return the Data Group (currently DG_IMAGE or a custom data group) for the upcoming transfer

There are five more DG_CONTROL operations for communications between the Source Manager and the Source. They are discussed in Chapter 5.

Image Operations from Application to Source

DG_IMAGE / DAT_CIECOLOR

MSG_GET :	Return the CIE XYZ information for the current transfer

DG_IMAGE / DAT_GRAYRESPONSE

MSG_RESET :	Reinstate identity response curve for grayscale data
MSG_SET :	Source uses specified response curve on grayscale data

DG_IMAGE / DAT_IMAGEFILEXFER

MSG_GET :	Initiate image acquisition using the Disk File transfer mode

DG_IMAGE / DAT_IMAGEINFO

MSG_GET :	Return information that describes the image for the next transfer

DG_IMAGE / DAT_IMAGELAYOUT

MSG_GET :	Describe physical layout / position of “original” image
MSG_GETDEFAULT :	Default information on the layout of the image
MSG_RESET :	Set layout information for the next transfer to defaults
MSG_SET :	Set layout for the next image transfer

DG_IMAGE / DAT_IMAGEMEMXFER

MSG_GET :	Initiate image acquisition using the Buffered Memory transfer mode

DG_IMAGE / DAT_IMAGENATIVEXFER

MSG_GET :	Initiate image acquisition using the Native transfer mode

DG_IMAGE / DAT_JPEGCOMPRESSION

MSG_GET :	Return JPEG compression parameters for current transfer
MSG_GETDEFAULT :	Return default JPEG compression parameters
MSG_RESET :	Use Source’s default JPEG parameters on JPEG transfers
MSG_SET :	Use specified JPEG parameters on JPEG transfers

DG_IMAGE / DAT_PALETTE8

MSG_GET :	Return palette information for current transfer
MSG_GETDEFAULT :	Return Source’s default palette information for current pixel type
MSG_RESET :	Use Source’s default palette for transfer of this pixel type
MSG_SET :	Use specified palette for transfers of this pixel type

DG_IMAGE / DAT_RGBRESPONSE

MSG_RESET :	Use Source’s default (identity) RGB response curve
MSG_SET :	Use specified response curve for RGB transfers

DG_AUDIO / DAT_AUDIOFILEXFER

MSG_GET :	Transfers audio data in file mode

DG_AUDIO / DAT_AUDIOINFO

MSG_GET :	Gets information about the current transfer

DG_AUDIO / DAT_AUDIONATIVEXFER

MSG_GET :	Transfers audio data in native mode

DSM_Entry Parameters

The parameters for the DG_xxxx, DAT_xxxx, and MSG_xxxx fields are determined by the operation triplet. The other parameters are filled as follows:

pOrigin

Refers to a copy of the application's TW_IDENTITY structure.

pDest

If the operation's destination is the Source Manager: Always holds a value of NULL. This indicates to the Source Manager that the operation is to be consumed by it not passed on to a Source.

If the operation's destination is a Source: This parameter references a copy of the Source's TW_IDENTITY structure that is maintained by the application. The application received this structure in response to the DG_CONTROL / DAT_IDENTITY / MSG_OPENDS operation sent from the application to the Source Manager. This is discussed more in the next section (Controlling a TWAIN Session from your Application - State 3 to 4).

pData

Always references a structure or variable corresponding to the TWAIN type specified by the DAT_xxxx parameter. Typically, but not always, the data argument type name corresponds to a TW_xxxx data structure name. For example, the DAT_IDENTITY argument type uses the corresponding TW_IDENTITY data structure. All data structures can be seen in the file called TWAIN.H. The application is responsible for allocating and deallocating the structure or element and assuring that pData correctly references it.

Note that there are two cases when the Source, rather than the application, allocates a structure that is used during an operation.

- One occurs during DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, and MSG_RESET operations. The application still allocates **pData* but the Source allocates a structure referenced by **pData* called a "container structure".
- The other occurs during the DG_IMAGE / DAT_JPEGCOMPRESSION operations. The topic of data compression is covered in Chapter 4.

In all cases, the application still deallocates all structures.

Controlling a TWAIN Session from Your Application

In addition to the preparations discussed at the beginning of this chapter, the application must be modified to actually initiate and control a TWAIN session.

The session consists of the seven states of the TWAIN protocol as introduced in the Technical Overview. However, the application is not forced to move the session from State 1 to State 7 without stopping. For example, some applications may choose to pause in State 3 and move among the higher states (4 - 7) to repeatedly open and close Sources when acquisitions are requested by the user. Another example of session flexibility occurs when an application transfers multiple images during a session. The application will repeatedly move the session from State 6 to State 7 then back to State 6 and forward to State 7 again to transfer the next image.

For the sake of simplicity, this chapter illustrates moving the session from State 1 to State 7 and then backing it out all the way from State 7 to State 1. The diagram on the next page shows the operation triplets that are used to transition the session from one state to the next. Detailed information about each state and its associated transitions follow. The topics include:

- Load the Source Manager and Get the DSM_Entry (State 1 to 2)
- Open the Source Manager (State 2 to 3)
- Select the Source (during State 3)
- Open the Source (State 3 to 4)
- Negotiate Capabilities with the Source (during State 4)
- Request the Acquisition of Data from the Source (State 4 to 5)
- Recognize that the Data Transfer is Ready (State 5 to 6)
- Start and Perform the Transfer (State 6 to 7)
- Conclude the Transfer (State 7 to 6 to 5)
- Disconnect the TWAIN Session (State 5 to 1 in sequence)

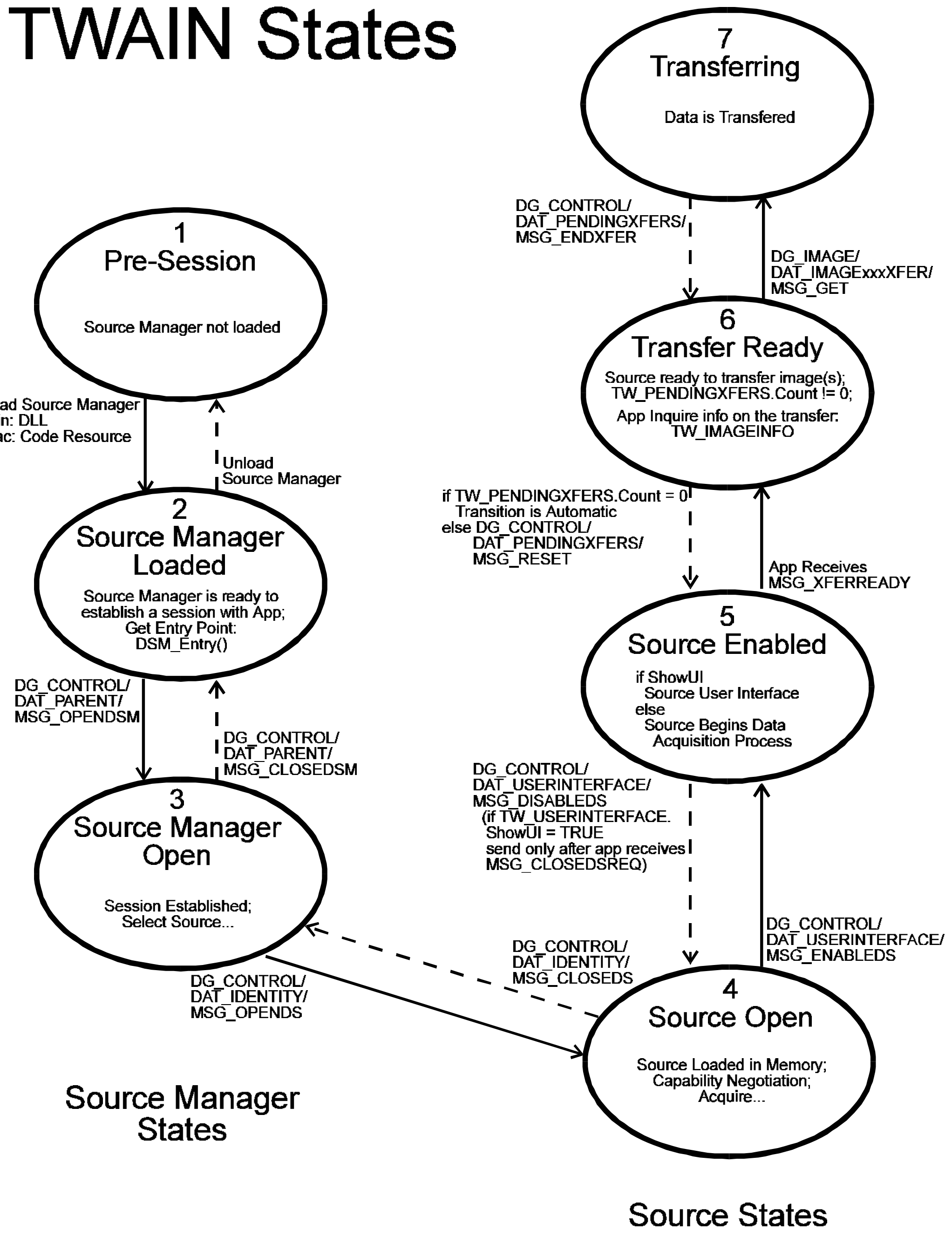


Figure 3-2. TWAIN States

State 1 to 2 - Load the Source Manager and Get the DSM_Entry

The application must load the Source Manager before it is able to call its DSM_Entry point.

Operations Used:

No TWAIN operations are used for this transition. Instead,

On Windows:

Load TWAIN.DLL using the LoadLibrary( ) routine.

Get the DSM_Entry by using the GetProcAddress( ) call.

On Macintosh:

Open the Source Manager code resource using OpenResFile( ) and GetResource( ).

Get the DSM_Entry by dereferencing the handle to the Source Manager.

On Windows:

The application can load the Source Manager by doing the following:

```
DSMENTRYPROC   pDSM_Entry;
HANDLE         hDSMLib;
char           szSMDir;
OFSTRUCT       of;
// check for the existence of the TWAIN_32.DLL file in the Windows
// directory
   GetWindowsDirectory (szSMDir, sizeof(szSMDir));
   /*** Could have been networked drive with trailing ‘\’ ***/
   if (szSMDir [(lstrlen (szSMDir) - 1)] != ‘\\’)
   {      lstrcat( szSMDir, "\\" );
   }

if ((OpenFile(szSMDir, &of, OF_EXIST) != -1)
{
   // load the DLL
   if (hDSMDLL = LoadLibrary(DSMName)) != NULL)
   {
      // check if library was loaded
      if (hDSMDLL >= (HANDLE)VALID_HANDLE)
      {
         if (lpDSM_Entry = (DSMENTRYPROC)GetProcAddress(hDSMDLL,
MAKEINTRESOURCE (1))) != NULL)
         {
            if (lpDSM_Entry )
               FreeLibrary(hDSMDLL);
         }
      }
   }
}
```

Note, the code appends TWAIN.DLL to the end of the Windows directory and verifies that the file exists before calling LoadLibrary( ). Applications are strongly urged to perform a dynamic run-time link to DSM_Entry( ) by calling LoadLibrary( ) rather than statically linking to TWAIN.LIB via the linker. If the TWAIN.DLL is not installed on the machine, Microsoft Windows will fail to load an application that statically links to TWAIN.LIB. If the Application has a dynamic link, however, it will be able to give users a meaningful error message, and perhaps continue with image acquisition facilities disabled.

After getting the DSM_Entry, the application must check pDSM_Entry. If it is NULL, it means that the Source Manager has not been installed on the user's machine and the application cannot provide any TWAIN services to the user. If NULL, the application must not attempt to call *pDSM_Entry as this would result in an Unrecoverable Application Error (UAE).

On Macintosh:

The Source Manager, named Source Manager, is a code resource on the Macintosh (its resource type is DSMR). Under System 6, this file resides within the TWAIN folder which lives in the System Folder. Under System 7, the TWAIN folder lives in the Preferences folder which lives in the System Folder. To access the Source Manager, the application must first load and lock down this code resource. The application must call the Source Manager (via the DSM_Entry() call) as a Pascal function.

The following code segments were written in Think "C" v5.0. Your exact coding may vary but the approach will be consistent.

The following code segment will load the Source Manager:

```
typedef PASCAL TW_UINT16(*DSMEntryFunc)(pTW_IDENTITY,
pTW_IDENTITY,
TW_UINT32,
TW_UINT16,
TW_UINT16,
TW_MEMREF);
DSMEntryFunc pDSM_Entry;
TW_INT16   SaveRes;

SaveRes=CurResFile();          /* save the current resource file  */

/* Be sure to change the working directory to the TWAIN folder    */
DSMRefNum=OpenResFile(DSMName); /* open the Source Manager file   */

DSMHandle=GetResource(DSMR_type,TWON_DSMCODEID); /* get the SM resource*/
HLock(DSMHandle);               /* lock the SM resource handle     */
pDSM_Entry=(pTW_INT16)*DSMHandle; /* get pointer to the DSM_Entry  */

/* Be sure to restore the working directory */
UseResFile(SaveRes);           /* restore current resource file */
```

After getting the DSM_Entry, application must check pDSM_Entry. If it is NULL, it means that the Source Manager has not been installed on the user's machine and the application cannot provide any TWAIN services to the user. If NULL, then application must not attempt to call *pDSM_Entry as this would result in a crash of the application.

State 2 to 3 - Open the Source Manager

The Source Manager has been loaded. The application must now open the Source Manager.

One Operation is Used:

DG_CONTROL / DAT_PARENT / MSG_OPENDSM

pOrigin

The application must allocate a structure of type TW_IDENTITY and fill in all fields except for the Id field. Once the structure is prepared, this pOrigin parameter should point at that structure.

During the MSG_OPENDSM operation, the Source Manager will fill in the Id field with a unique identifier of the application. The value of this identifier is only valid while the application is connected to the Source Manager.

The application must save the entire structure. From now on, the structure will be referred to by the pOrigin parameter to identify the application in every call the application makes to DSM_Entry( ).

The TW_IDENTITY structure is defined in the TWAIN.H file but for quick reference, it looks like this:

```
/* DAT_IDENTITY Identifies the program/library/code */
/*              resource.                           */
typedef struct {
   TW_UINT32    Id; /* Unique number for identification*/
   TW_VERSION   Version;
   TW_UINT16    ProtocolMajor;
   TW_UINT16    ProtocolMinor;
   TW_UINT32    SupportedGroups;/*Bit field OR combination */
                                /*of DG_constants found in */
                                /*the TWAIN.H file         */
   TW_STR32     Manufacturer;
   TW_STR32     ProductFamily;
   TW_STR32     ProductName;
} TW_IDENTITY, FAR *pTW_IDENTITY;
```

pDest

Set to NULL indicating the operation is intended for the Source Manager.

pData

Typically, you would expect to see this point to a structure of type TW_PARENT but this is not the case. This is an exception to the usual situation where the DAT field of the triplet identifies the data structure for pData.

- **On Windows:** pData points to the window handle (hWnd) that will act as the Source's “parent”. The variable is of type TW_INT32. For 16 bit Microsoft Windows, the handle is stored in the low word of the 32 bit integer and the upper word is set to zero. If running under the WIN32 environment, this is a 32 bit window handle. The Source Manager will maintain a copy of this window handle for posting messages back to the application.
- **On Macintosh:** pData should be a 32-bit NULL value.

How to Initialize the TW_IDENTITY Structure

Here is a Windows example of code used to initialize the application's TW_IDENTITY structure.

```
TW_IDENTITY     AppID;           // App's identity structure
   AppID.Id = 0;                 // Initialize to 0 (Source Manager
                                 // will assign real value)
   AppID.Version.MajorNum = 3;   //Your app's version number
   AppID.Version.MinorNum = 5;
   AppID.Version.Language = TWLG_ENGLISH_USA;
   AppID.Version.Country  = TWCY_USA;
   lstrcpy (AppID.Version.Info, "Your App's Version String");
   AppID.ProtocolMajor = TWON_PROTOCOLMAJOR;
   AppID.ProtocolMinor = TWON_PROTOCOLMINOR;
   AppID.SupportedGroups = DG_IMAGE | DG_CONTROL;
   lstrcpy (AppID.Manufacturer, "App's Manufacturer");
   lstrcpy (AppID.ProductFamily, "App's Product Family");
   lstrcpy (AppID.ProductName, "Specific App Product Name");
```

On Windows: Using DSM_Entry to open the Source Manager

```
TW_UINT16  rc;
rc = (*pDSM_Entry) (&AppID,
                    NULL,
                    DG_CONTROL,
                    DAT_PARENT,
                    MSG_OPENDSM,
                    (TW_MEMREF) &hWnd);
```

where AppID is the TW_IDENTITY structure that the application set up to identify itself and hWnd is the application's main window handle.

On Macintosh: Using DSM_Entry to open the Source Manager

```
TW_UINT16  SaveRes;

SaveRes=CurResFile();  /* Save the current resource file */
UseResFile(DSMRefNum); /* Set Source Manager resource file as
                       /* current*/

rc = (*pDSM_Entry) (&AppID,
                    NULL,
                    DG_CONTROL,
                    DAT_PARENT,
                    MSG_OPENDSM,
                    NULL);

UseResFile(SaveRes);   /* Restore the resource file */
```

where AppID is the TW_IDENTITY structure that the application set up to identify itself. The same approach is used by the application to call the DSM_Entry( ) function throughout the remainder of the TWAIN session.

Note: Whenever your application uses a TWAIN operation, **always check the Return Code** sent back through the DSM_Entry function to be certain an operation is successful. If an operation indicates failure, use the DG_CONTROL / DAT_STATUS / MSG_GET operation to get the Condition Code that indicates the cause of failure. The application identifies who the status triplet be sent to, the Source Manager or Source, depending on which one received the original operation that failed.

State 3 - Select the Source

The Source Manager has just been opened and is now available to assist your application in the selection of the desired Source.

One Operation is Used:

DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

pOrigin

Points to the application's TW_IDENTITY structure. The desired data type should be specified by the application. This was done when you initialized the SupportedGroups field in your application's TW_IDENTITY structure.

This causes the Source Manager to make available for selection by the user only those Sources that can provide the requested data type(s). All other Sources are grayed out. (Note, if more than one data type were available, for example image and text, and the application wanted to accept both types of data, it would do a bit-wise OR of the types' constants and place the results into the SupportedGroups field.)

pDest

Set to NULL.

pData

Points to a structure of type TW_IDENTITY. The application must allocate this structure prior to making the call to DSM_Entry. Once the structure is allocated, the application must:

- Set the Id field to zero.
- Set the ProductName field to the null string ("\0"). (If the application wants a specific Source to be highlighted in the Select Source dialog box, other than the system default, it can enter the ProductName of that Source into the ProductName field instead of null. The system default Source and other available Sources can be determined by using the DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT, MSG_GETFIRST and MSG_GETNEXT operations.)

Additional fields of the structure will be filled in by the Source Manager during this operation to identify the selected Source. Make sure the application keeps a copy of this updated structure after completing this call. You will use it to identify the Source from now on.

The most common approach for selecting the Source is to use the Source Manager's Select Source dialog box. This is typically displayed when the user clicks on your Select Source option. To do this:

1. The application sends a DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT operation to the Source Manager to have it display its dialog box. The dialog displays a list of all Sources that are installed on the system that can provide data of the type specified by the application. It highlights the Source that is the system default unless the application requests otherwise.
2. The user selects a Source or presses the Cancel button. If no devices are available, the Select Source Dialog's Select/OK button will be grayed out and the user will have no choice but to select Cancel.
3. The application must check the Return Code of DSM_Entry to determine the user's action.
   a. **If TWRC_SUCCESS**: Their selected Source is listed in the TW_IDENTITY structure pointed to by the pData parameter and is now the default Source.
   b. **If TWRC_CANCEL**: The user either clicked Cancel intentionally or had no other choice because no devices were listed. Do not attempt to open a Source.
   c. **If TWRC_FAILURE**: Use the DG_CONTROL / DAT_STATUS / MSG_GET operation (sent to the Source Manager) to determine the cause. The most likely cause is a lack of sufficient memory.

As an alternative to using the Source Manager's Select Source dialog, the application can devise its own method for selecting a Source. For example, it could create and display its own user interface or simply select a Source without offering the user a choice. This alternative is discussed in Chapter 4.

State 3 to 4 - Open the Source

The Source Manager is open and able to help your application open a Source.

One Operation is Used:

DG_CONTROL / DAT_IDENTITY / MSG_OPENDS

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Set to NULL.

pData

Points to a structure of type TW_IDENTITY.

Typically, this points to the application's copy of the Source's TW_IDENTITY structure filled in during the MSG_USERSELECT operation previously.

However, if the application wishes to have the Source Manager simply open the default Source, it can do this by setting the TW_IDENTITY.ProductName field to "\0" (null string) and the TW_IDENTITY.Id field to zero.

During the MSG_OPENDS operation, the Source Manager assigns a unique identifier to the Source and records it in the TW_IDENTITY.Id field. Copy the resulting TW_IDENTITY structure. Once the Source is opened, the application will point to this resulting structure via the pDest parameter on every call that the application makes to DSM_Entry where the desired destination is this Source.

Note: The user is not required to take advantage of the Select Source option. They may click on the Acquire option without having selected a Source. In that case, your application should open the default Source. **The default source is either the last one used by the user or the last one installed.**

State 4 - Negotiate Capabilities with the Source

At this point, the application has a structure identifying the open Source. Operations can now be directed from the application to that Source. To receive a single image from the Source, only one capability, CAP_XFERCOUNT, must be negotiated now. All other capability negotiation is optional.

Two Operations are Used:

DG_CONTROL / DAT_CAPABILITY / MSG_GET

DG_CONTROL / DAT_CAPABILITY / MSG_SET

The parameters for each of the operations, in addition to the triplet, are these:

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Points to the desired Source's TW_IDENTITY structure. The Source Manager will receive the DSM_Entry call, recognize that the destination is a Source rather than itself, and pass the operation along to the Source via the DS_Entry function.

pData

Points to a structure of type TW_CAPABILITY.

The definition of TW_CAPABILITY is:

```
typedef struct {
  TW_UINT16   Cap;       /* ID of capability to get or set */
  TW_UINT16   ConType;   /* TWON_ONEVALUE, TWON_RANGE,     */
                         /* TWON_ENUMERATION or TWON_ARRAY */
  TW_HANDLE  hContainer; /* Handle to container of type    */
                         /* ConType                        */
} TW_CAPABILITY, FAR *pTW_CAPABILITY;
```

The Source allocates the container structure pointed to by the hContainer field when called by the MSG_GET operation. The application allocates it when calling with the MSG_SET operation. Regardless of who allocated it, the application deallocates the structure either when the operation is complete or when the application no longer needs to maintain the information.

Each operation serves a special purpose:

MSG_GET

Since Sources are not required to support all capabilities, this operation can be used to determine if a particular TWAIN-defined capability is supported by a Source. The application needs to set the Cap field of the TW_CAPABILITY structure to the identifier representing the capability of interest. The constants identifying each capability are listed in the TWAIN.H file.

If the capability is supported and the operation is successful, it returns the Current, Default, and Available values. These values reflect previous MSG_SET operations on the capability which may have altered them from the TWAIN default values for the capability.

This operation may fail due to several causes. If the capability is not supported by the Source, the Return Code will be TWRC_FAILURE and the condition code will be one of the following:

TWCC_CAPUNSUPPORTED	Capability not supported by Source
TWCC_CAPBADOPERATION	Operation not supported by capability
TWCC_CAPSEQERROR	Capability has dependency on other capability

Applications should be prepared to receive the condition code TWCC_BADCAP from Sources written prior to TWAIN 1.7, which maps to any of the three situations mentioned above.

MSG_SET

Changes the Current or Available Value(s) of the specified capability to those requested by the application. The application may choose to set just the capability's Current Value or it may specify a list of values for the Source to use as the complete set of Available Values for that capability.

Note: Source is not required to limit values based on the application's request although it is strongly recommended that they do so. If the Return Code indicates TWRC_FAILURE, check the Condition Code. A code of TWCC_BADVALUE can mean:

- The application sent an invalid value for this Source's range.
- The Source does not allow the setting of this capability.
- The Source doesn't allow the type of container used by the application to set this capability.

Capability negotiation gives the application developer power to guide the Source and control the images they receive from the Source. The negotiation typically occurs during State 4. The following material illustrates only one very basic capability and container structure. Refer to Chapter 4 for a more extensive discussion of capabilities including information on how to delay the negotiation of some capabilities beyond State 4.

Note: It is important here to once again remind application writers to always check the return code from any negotiated capabilities transactions.

Set the Capability to Specify the Number of Images the Application can Transfer

The capability that specifies how many images an application can receive during a TWAIN session is CAP_XFERCOUNT. All Sources must support this capability. Possible values for CAP_XFERCOUNT are:

Value:	**Description:**
1	Application wants to receive a single image.
greater than 1	Application wants to receive this specific number of images.
-1	Application can accept any arbitrary number of images during the session. **This is the default for this capability.**
0	This value has no legitimate meaning and the application should not set the capability to this value. If a Source receives this value during a MSG_SET operation, it should maintain the Current Value without change and return TWRC_FAILURE and TWCC_BADVALUE.

The default value allows multiple images to be transferred. The following is a simple code example illustrating the setting of a capability and specifically showing how to limit the number of images to one. Notice there are differences between the code for Windows and Macintosh applications. Both versions are included here with ifdef statements for MSWIN versus MAC.

```
TW_CAPABILITY   twCapability;
TW_INT16              count;
TW_STATUS             twStatus;
TW_UINT16             rc;
#ifdef _MSWIN_
pTW_ONEVALUE   pval;
#endif
#ifdef _MAC_
TW_HANDLE      h;
pTW_INT16      pInt16;
#endif

//-----Setup for MSG_SET for CAP_XFERCOUNT
twCapability.Cap = CAP_XFERCOUNT;
twCapability.ConType = TWON_ONEVALUE;

#ifdef _MSWIN_
twCapability.hContainer = GlobalAlloc(GHND, sizeof(TW_ONEVALUE));
pval = (pTW_ONEVALUE) GlobalLock(twCapability.hContainer);
pval->ItemType = TWTY_INT16;
pval->Item = 1;         //This app will only accept 1 image
GlobalUnlock(twCapability.hContainer);
#endif

#ifdef _MAC_
twCapability.hContainer = (TW_HANDLE)h = NewHandle(sizeof(TW_ONEVALUE));
((TW_ONEVALUE*)(*h))->ItemType = TWTY_INT16;
count = 1;         //This app will only accept 1 image
pInt16 = ((TW_ONEVALUE*)(*h))->Item;
*pInt16 = count;
#endif
```

```
//-----Set the CAP_XFERCOUNT
rc = (*pDSM_Entry) (&AppID,
                    &SourceID,
                    DG_CONTROL,
                    DAT_CAPABILITY,
                    MSG_SET,
                   (TW_MEMREF)&twCapability);
#ifdef _MSWIN_
GlobalFree((HANDLE)twContainer.hContainer);
#endif
#ifdef _MAC_
DisposHandle((HANDLE)twContainer.hContainer);
#endif
//-----Check Return Codes
//SUCCESS
if (rc == TWRC_SUCCESS)
    //the value was set
//APPROXIMATION MADE
else if (rc == TWRC_CHECKSTATUS)
    {
    //The value could not be matched exactly
    //MSG_GET to get the new current value
    twCapability.Cap = CAP_XFERCOUNT;
    twCapability.ConType = TWON_DONTCARE16;   //Source will specify
    twCapability.hContainer = NULL; //Source allocates and fills
container
    rc = (*pDSM_Entry) (&AppID,
                &SourceID,
                DG_CONTROL,
                DAT_CAPABILITY,
                MSG_GET,
               (TW_MEMREF)&twCapability);
    //remember current value
    #ifdef _MSWIN_
    pval = (pTW_ONEVALUE) GlobalLock(twCapability.hContainer);
    count = pval->Item;
    //free hContainer allocated by Source
    GlobalFree((HANDLE)twCapability.hContainer);
    #endif
    #ifdef _MAC_
    pInt16 = ((TW_ONEVALUE*)(*h))->Item;
    count = *pInt16;
    //free hContainer allocated by Source
    DisposeHandle((HANDLE)twCapability.hContainer);
    #endif
    }
```

```
//MSG_SET FAILED
else if (rc == TWRC_FAILURE)
    {
    //check Condition Code
    rc = (*pDSM_Entry) (&AppID,
                        &SourceID,
                        DG_CONTROL,
                        DAT_STATUS,
                        MSG_GET,
                       (TW_MEMREF)&twStatus);
    switch (twStatus.ConditionCode)
        {
        TWCC_BADCAP:
        TWCC_CAPUNSUPPORTED:
        TWCC_CAPBADOPERATION:
        TWCC_CAPSEQERROR:
               //Source does not support setting this cap
               //All Sources must support CAP_XFERCOUNT
               break;
        TWCC_BADDEST:
               //The Source specified by pSourceID is not open
               break;
        TWCC_BADVALUE:
               //The value set was out of range for this Source
               //Use MSG_GET to determine what setting was made
               //See the TWRC_CHECKSTATUS case handled earlier
               break;
        TWCC_SEQERROR:
               //Operation invoked in invalid state
               break;
       }
    }
```

Other Capabilities

Image Type

Although not shown, the application should be aware of the Source’s ICAP_PIXELTYPE and ICAP_BITDEPTH. If your application cannot accept all of the Source’s Available Values, capability negotiation should be done. (Refer to Chapter 4.)

Transfer Mode

The default transfer mode is Native. That means the Source will access the largest block of memory available and use it to transfer the entire image to the application at once. If the available memory is not large enough for the transfer, then the Source should fail the transfer. The application does not need to do anything to select this transfer mode. If the application wishes to specify a different transfer mode, Disk File or Buffered Memory, further capability negotiation is required. (Refer to Chapter 4.)

State 4 to 5 - Request the Acquisition of Data from the Source

The Source device is open and capabilities have been negotiated. The application now enables the Source so it can show its user interface, if requested, and prepare to acquire data.

One Operation is Used:

DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Points to the Source's TW_IDENTITY structure.

pData

Points to a structure of type TW_USERINTERFACE.

The definition of TW_USERINTERFACE is:

```
typedef struct {
  TW_BOOL    ShowUI;
  TW_BOOL    ModalUI;
  TW_HANDLE  hParent;
} TW_USERINTERFACE, FAR *pTW_USERINTERFACE;
```

Set the ShowUI field to TRUE if you want the Source to display its user interface. Otherwise, set to FALSE.

The Source will set the ModalUI field to TRUE if its user interface is modal. If the interface is modeless, the field is set to FALSE.

The application sets the hParent field differently depending on the platform on which the application runs.

- **On Windows -** The application should place a handle to the Window that is acting as the Source's parent.
- **On Macintosh -** The application sets this field to NULL.

In response to the user choosing the application's Acquire menu option, the application sends this operation to the Source to enable it. The application typically requests that the Source display the Source's user interface to assist the user in acquiring data. If the Source is told to display its user interface, it will display it when it receives the operation triplet and it will set the ModalUI field of the data structure appropriately. Modal and Modeless interfaces are discussed in Chapters 4 and 5. Sources **must** check the `ShowUI` field and return an error if they cannot support the specified mode. In other words it is unacceptable for a source to ignore a `ShowUI = FALSE` request and still activate its user interface. The application may develop its own user interface instead of using the Source's. This is discussed in Chapter 4.

Note: Once the Source is enabled via the DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS operation, all events that enter the application's main event loop must be immediately forwarded to the Source. The explanation for this was given earlier in this chapter when you were instructed to modify the event loop in preparation for a TWAIN session.

State 5 to 6 - Recognize that the Data Transfer is Ready

The Source is now working with the user to arrange the transfer of the desired data. Unlike all the earlier transitions, the Source, not the application, controls the transition from State 5 to State 6.

No Operations (from the application) are Used:

This transition is not triggered by the application sending an operation. The Source causes the transition.

Remember while the Source is enabled, the application is forwarding all events in its event loop to the Source by using the DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation. The TW_EVENT data structure associated with this operation looks like this:

```
typedef struct {
  TW_MEMREF  pEvent;   /*Windows pMSG or MAC pEvent                   */
  TW_UINT16  TWMessage;/*TW message from the Source to the application*/
} TW_EVENT, FAR *pTW_EVENT;
```

The Source can set the TWMessage field to signal when the Source is ready to transfer data. Following each DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation, the application must check the TWMessage field. If it contains MSG_XFERREADY, the session is in State 6 and the Source will wait for the application to request the actual transfer of data.

State 6 to 7 - Start and Perform the Transfer

The Source indicated it is ready to transfer data. It is waiting for the application to inquire about the image details, initiate the actual transfer, and, hence, transition the session from State 6 to 7. If the initiation (DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET) fails, the session does not transition to State 7 but remains in State 6.

Two Operations are Used:

DG_IMAGE / DAT_IMAGEINFO / MSG_GET

pOrigin

Points to the application’s TW_IDENTITY structure.

pDest

Points to the Source’s TW_IDENTITY structure.

pData

Points to a structure of type TW_IMAGEINFO. The definition of TW_IMAGEINFO is:

```
typedef struct {
   TW_FIX32    XResolution;
   TW_FIX32    YResolution;
   TW_INT32    ImageWidth;
   TW_INT32    ImageLength;
   TW_INT16    SamplesPerPixel;
   TW_INT16    BitsPerSample[8];
   TW_INT16    BitsPerPixel;
   TW_BOOL     Planar;
   TW_INT16    PixelType;
   TW_UINT32   Compression;
}  TW_IMAGEINFO, FAR *pTW_IMAGEINFO;
```

The Source will fill in information about the image that is to be transferred. The application uses this operation to get the information regardless of which transfer mode (Native, Disk File, or Buffered Memory) will be used to transfer the data.

DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Points to the Source's TW_IDENTITY structure.

pData

Points to a TW_UINT32 variable. This is an exception from the typical pattern.

- **On Windows:** This is a pointer to a handle variable. For 16 bit Microsoft Windows, the handle is stored in the low word of the 32-bit integer and the upper word is set to zero. If running under the WIN32 environment, this is a 32 bit window handle. The Source will set pHandle to point to a device-independent bitmap (DIB) that it allocates.
- **On Macintosh:** This is a pointer to a PicHandle. The Source will set pHandle to point to a PicHandle that the Source allocates.

In either case, the application is responsible for deallocating the memory block holding the Native-format image.

The application may want to inquire about the image data that it will be receiving. The DG_IMAGE / DAT_IMAGEINFO / MSG_GET operation allows this. Other operations, such as DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET, provide additional information. This information can be used to determine if the application actually wants to initiate the transfer.

To actually transfer the data in the Native mode, the application invokes the DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET operation. The Native mode is the default transfer mode and will be used unless a different mode was negotiated via capabilities in State 4. For the Native mode transfer, the application only invokes this operation once per image. The Source returns the TWRC_XFERDONE value when the transfer is complete. This type of transfer cannot be aborted by the application once initiated. (Whether it can be aborted from the Source's User Interface depends on the Source.) Use of the other transfer modes, Disk File and Buffered Memory, are discussed in Chapter 4.

The following code illustrates how to get information about the image that will be transferred and how to actually perform the transfer. This code segment is continued in the next section (State 7 to 6 to 5).

```
// After receiving MSG_XFERREADY
TW_UINT16 TransferNativeImage()
{
      TW_IMAGEINFO   twImageInfo;
      TW_UINT16      rc;
      TW_UINT32      hBitmap;
      TW_BOOL        PendingXfers = TRUE;
while (PendingXfers)
{
      rc = (*pDSM_Entry)(&AppId,
                          &SourceId,
                          DG_IMAGE,
                          DAT_IMAGEINFO,
                          MSG_GET,
                         (TW_MEMREF)&twImageInfo);
      if (rc == TWRC_SUCCESS)
             Examine the image information
             // Transfer the image natively
             hBitmap = NULL;
             rc = (*pDSM_Entry)(&AppId,
                                 SourceId,
                                 DG_IMAGE,
                                 DAT_IMAGENATIVEXFER,
                                 MSG_GET,
                                (TW_MEMREF)&HbITMAP);
             // Check the return code
             switch(rc)
             {
                    case TWRC_XFERDONE:
      // Notes: hBitmap points to a valid image Native image (DIB or
      // PICT)
      // The application is now responsible for deallocating the memory.
      // The source is currently in state 7.
      // The application must now acknowledge the end of the transfer,
      // determine if other transfers are pending and shut down the data
      // source.
                    PendingXfers = DoEndXfer();  //Function found in code
                                                 //example in next section
                           break;
```

```
                case TWRC_CANCEL:
    // The user canceled the transfer.
    // hBitmap is an invalid handle but memory was allocated.
    // Application is responsible for deallocating the memory.
    // The source is still in state 7.
    // The application must check for pending transfers and shut down
    // the data source.
                PendingXfers = DoEndXfer();  //Function found in code
                                            //example in next section
                       break;
                case TWRC_FAILURE:
    // The transfer failed for some reason.
    // hBitmap is invalid and no memory was allocated.
    // Condition code will contain more information as to the cause of
    // the failure.
    // The state transition failed, the source is in state 6.
    // The image data is still pending.
    // The application should abort the transfer.
                   DoAbortXfer(MSG_RESET); //Function in next section
                   PendingXfers = FALSE;
                   break;
          }
    }
}
//Check the return code
switch (rc)
       {
       case TWRC_XFERDONE:
             //hBitMap points to a valid Native Image (DIB or PICT)
             //The application is responsible for deallocating the memory
             //The source is in State 7
             //Acknowledge the end of the transfer
                    goto LABEL_DO_ENDXFER //found in next section
              break;
       case TWRC_CANCEL:
              //The user canceled the transfer
              //hBitMap is invalid
              //The source is in State 7
              //Acknowledge the end of the transfer
                    goto LABEL_DO_ENDXFER //found in next section
              break;
       case TWRC_FAILURE:
              //The transfer failed
              //hBitMap is invalid and no memory was allocated
              //Check Condition Code for more information
              //The state transition failed, the source is in State 6
              //The image data is still pending
              //To abort the transfer
                    goto LABEL_DO_ENDXFER //found in code example for
                                          //the next section
              break;
   }
```

State 7 to 6 to 5 - Conclude the Transfer

While the transfer occurs, the session is in State 7. When the Source indicates via the Return Code that the transfer is done (TWRC_XFERDONE) or canceled (TWRC_CANCEL), the application needs to transition the session backwards.

One Operation is Used:

DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Points to the Source's TW_IDENTITY structure.

pData

Points to a structure of type TW_PENDINGXFERS.

The definition of TW_PENDINGXFERS is:

```
typedef struct {
  TW_UINT16  Count;
  TW_UINT32  Reserved;
} TW_PENDINGXFERS, FAR *pTW_PENDINGXFERS;
```

The DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation is sent by the application to the Source at the end of every transfer, successful or canceled, to indicate the application has received all the data it expected.

After this operation returns, the application should examine the pData->Count field to determine if there are more images waiting to be transferred. The value of pData->Count indicates the following:

Value	Description
pData->Count = 0	If zero, the Source will "automatically" transition back to State 5 without the application needing to take any additional action. **Application writers please make special note of this instance of an implied source transition.** The application should return to its main event loop and await notification from the Source (either MSG_XFERREADY or MSG_CLOSEDSREQ).
pData->Count = -1 or pData->Count > 0	The Source has more transfers available and is waiting in State 6. If the value is -1, that means the Source has another image available but it is unsure of how many more will be available. This might occur if the Source was controlling a device equipped with a document feeder and some unknown number of documents were stacked in that feeder. If the number of images is known, the Count will be a value greater than 0. Either way, the Source will remain in State 6 ready for the application to initiate another transfer. The Source will **NOT** send another MSG_XFERREADY to trigger this. The application should proceed as if it just received a MSG_XFERREADY.

If more images were pending and your application does not wish to transfer all of them, you can discard one or all pending images by doing the following:

- **To discard just the next pending image**, use the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation. Then, check the Count field again to determine if there are additional images pending.
- **To discard all pending images**, use the DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET operation. Following successful execution of this operation, the session will be in State 5.

The following code is a continuation of the code example started in the State 6 to 7 section. It illustrates how to conclude the transfer.

```
void DoEndXfer()
{
   TW_PENDINGXFERS      twPendingXfers;

   // If the return code from DG_IMAGE/DAT_IMAGENATIVEXFER/MSG_GET was
   // TWRC_CANCEL or TWRC_DONE

   // Acknowledge the end of the transfer
   rc = (*pDSM_Entry)(&AppId,
                                SourceId,
                                DG_CONTROL,
                                DAT_PENDINGXFERS,
                                MSG_ENDXFER,
                                (TW_MEMREF)&twPendingXfers);

   if (rc == TWRC_SUCCESS)
   {
          // Check for additional pending xfers
          if (twPendingXfers.Count == 0)
          {
                  // Source is now in state 5. NOTE THE IMPLIED STATE
                  // TRANSITION! Disable and close the source and
                  // return to TransferNativeImage with a FALSE notifying
                  // it to not attempt further image transfers.

                  DisableAndCloseDS();
                  return(FALSE);
          }
          else
          {
                  // Source is in state 6 ready to transfer another image
                  if want to transfer this image
                  {
                         // returns to the caller, TransferNativeImage
                         // and allows the next image to transfer

                         return TRUE;
                  }
                  else if want to abort and skip over this transfer
                  {
                         // The current image will be skipped, and the
                         // next, if exists will be acquired by returning
                         // to TransferNativeImage
                         if (DoAbortXfer(MSG_ENDXFER) > 0)
                                return(TRUE);
                         else
                                return(FALSE);
                  }
                                }
          }
   }
}
TW_UINT16 DoAbortXfer(TW_UINT16 AbortType)
{
      rc = (*pDSM_Entry)(&AppId,
                               SourceId,
                               DG_CONTROL,
                               DAT_PENDINGXFERS,
                               MSG_ENDXFER,
                              (TW_MEMREF)&twPendingXfers);
      if (rc == TWRC_SUCCESS)
      {
             // If the next image is to be skipped, but subsequent images
             // are still to be acquired, the PendingXfers will receive
             // the MSG_ENDXFER, otherwise, PendingXfers will receive
             // MSG_RESET.
             rc = (*pDSM_Entry)(&AppId,
                                    SourceId,
                                    DG_CONTROL,
                                    DAT_PENDINGXFERS,
                                    AbortType,
                                   (TW_MEMREF)&twPendingXfers);
      }
}
//To abort all pending transfers:
LABEL_ABORT_ALL:
             {
                 rc = (*pDSM_Entry) (&AppID,
                        &SourceID,
                        DG_CONTROL,
                        DAT_PENDINGXFERS,
                        MSG_RESET,
                       (TW_MEMREF)&twPendingXfers);
                 if (rc == TWRC_SUCCESS)
                        //Source is now in state 5
      }
}
```

State 5 to 1 - Disconnect the TWAIN Session

Once the application has acquired all desired data from the Source, the application can disconnect the TWAIN session. To do this, the application transitions the session backwards.

In the last section, the Source transitioned to State 5 when there were no more images to transfer (TW_PENDINGXFERS.Count = 0) or the application called the DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET operation to purge all remaining transfers. To back out the remainder of the session:

Three Operations (plus some platform-dependent code) are Used:

To move from State 5 to State 4

DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Points to the Source's TW_IDENTITY structure.

pData

Points to a structure of type TW_USERINTERFACE.

The definition of TW_USERINTERFACE is:

```
typedef struct {
  TW_BOOL    ShowUI;
  TW_BOOL    ModalUI;
  TW_HANDLE  hParent;
} TW_USERINTERFACE, FAR *pTW_USERINTERFACE;
```

Its contents are not used.

Note the following:

- **If the Source's User Interface was displayed:** This operation causes the Source's user interface, if displayed during the transition from State 4 to 5, to be lowered. This operation is sent by the application in response to a MSG_CLOSEDSREQ from the Source. This request from the Source appears in the TWMessage field of the TW_EVENT structure. It is sent back from the DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation used by the application to send events to the application.
- **If the application did not have the Source's User Interface displayed:** The application invokes this command when all transfers have been completed. In addition, the application could invoke this operation to transition back to State 4 if it wanted to modify one or more of the capability settings before acquiring more data.

To move from State 4 to State 3

DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Should reference a NULL value (indicates destination is Source Manager)

pData

Points to a structure of type TW_IDENTITY

This is the same TW_IDENTITY structure that you have used throughout the session to direct operation triplets to this Source.

When this operation is completed, the Source is closed. (In a more complicated scenario, if the application had more than one Source open, it must close them all before closing the Source Manager. Once all Sources are closed and the application does not plan to initiate any other TWAIN session with another Source, the Source Manager should be closed by the application.)

To move from State 3 to State 2

DG_CONTROL / DAT_PARENT / MSG_CLOSEDSM

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Should reference a NULL value (indicates destination is Source Manager)

pData

Typically, you would expect to see this point to a structure of type TW_PARENT but this is not the case. This is an exception to the usual situation where the DAT field of the triplet identifies the data structure for pData.

On Windows: pData points to the window handle (hWnd) that acted as the Source's "parent". The variable is of type TW_INT32. For 16 bit Microsoft Windows, the handle is stored in the low word of the 32 bit integer and the upper word is set to zero. If running under the WIN32 environment, this is a 32 bit window handle.

On Macintosh: pData should be a 32-bit NULL value.

To Move from State 2 to State 1

Once the Source Manager has been closed, the application must unload the DLL (on Windows) or code resource (on Macintosh) from memory before continuing.

On Windows:

Use FreeLibrary(hDSMLib); where hDSMLib is the handle to the Source Manager DLL returned from the call to LoadLibrary( ) seen earlier (in the State 1 to 2 section).

On Macintosh:

```
HUnlock(DSMHandle);  /* unlock the handle to the Source Manager */
ReleaseResource(DSMHandle);  /* release the Source Manager      */
CloseResFile(DSMRefNum);     /* close the Source Manager file   */
```

TWAIN Session Review

Applications have flexibility regarding which state they leave their TWAIN sessions in between TWAIN commands (such as Select Source and Acquire).

For example:

- An application might load the Source Manager on start-up and unload it on exit. Or, it might load the Source Manager only when it is needed (as indicated by Select Source and Acquire).
- An application might open a Source and leave it in State 4 between acquires.

The following is the simplest view of application's TWAIN flow. All TWAIN actions are initiated by a TWAIN command, either user-initiated (Select Source and Acquire) or notification from the Source (MSG_XFERREADY and MSG_CLOSEDSREQ).

Application Receives	**State**	**Application Action**
Select Source...	1 -> 2	Load Source Manager
	2 -> 3	DG_CONTROL / DAT_PARENT / MSG_OPENDSM
		DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT
	3 -> 2	DG_CONTROL / DAT_PARENT / MSG_CLOSEDSM
	2 -> 1	Unload Source Manager
Acquire...	1 -> 2	Load Source Manager
	2 -> 3	DG_CONTROL / DAT_PARENT / MSG_OPENDSM
	3 -> 4	DG_CONTROL / DAT_IDENTITY / MSG_OPENDS
		Capability Negotiation
	4 -> 5	DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS
MSG_XFERREADY	6	For each pending transfer:
		DG_IMAGE / DAT_IMAGEINFO / MSG_GET
		DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
		DG_CONTROL / DAT_CAPABILITY / MSG_GETCURRENT
	6 -> 7	DG_IMAGE / DAT_IMAGExxxxXFER / MSG_GET
	7 -> 6	DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER
	6 -> 5	Automatic transition to State 5 if TW_PENDINGXFERS.Count equals 0.
MSG_CLOSEDSREQ	5 -> 4	DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS
	4 -> 3	DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS
	3 -> 2	DG_CONTROL / DAT_PARENT / MSG_CLOSEDSM
	2 -> 1	Unload the Source Manager

Error Handling

Your application must be robust enough to recognize and handle error conditions that may occur during a TWAIN session. Every TWAIN operation triplet has a defined set of Return Codes and Conditions Codes that it may generate. These codes are listed on the reference pages for each triplet located in Chapter 7. Be sure to check the Return Code following every call to the DSM_Entry function. If it is TWRC_FAILURE, make sure your code checks the Condition Code and handles the error condition appropriately.

The following code segment illustrates the basic operations for doing this:

```
TW_STATUS    twStatus;

if (rc == TWRC_FAILURE)
       //check Condition Code
       rc = (*pDSM_Entry) (&AppID,
              &SourceID,
              DG_CONTROL,
              DAT_STATUS,
              MSG_GET,
             (TW_MEMREF)&twStatus);
       switch (twStatus.ConditionCode)
              //handle each possible Condition Code for the operation
```

Common Types of Error Conditions

Sequence Errors

The TWAIN protocol allows the invoking of specific operations only while the TWAIN session is in a particular state or states. The valid states for each operation are listed on the operation's reference pages in Chapter 7. If an operation is called from an inappropriate state, the call will return an error, TWRC_FAILURE, and set the Condition Code to TWCC_SEQERROR. Although this error should not occur if both the application and Source are behaving correctly, it is possible for the session to get out of sync.

If this error occurs, correct it by assuming the Source believes it is in State 7. The application should invoke the correct operations to back up from State 7 to State 6 and so on down the states until an operation succeeds. Then, the application can continue or terminate the session.

The following pseudo code illustrates this:

```
if (TWCC_SEQERROR)
      //  Assume State 7, start backing out from State 7 until
      //  the Condition Code != TWCC_SEQERROR
      State 7 to 6    DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER
      State 6 to 5    DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET
      State 5 to 4    DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS
      State 4 to 3    DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS
```

Low Memory Errors

Another common type of error condition occurs when insufficient memory is available to perform a requested operation. The most likely times for this to occur are:

- When a Source is being opened
- When a Source is being enabled
- During a Native image transfer

Your application must check the Return Code and Condition Code (TWRC_FAILURE / TWCC_LOWMEMORY) to recognize this. Your application may be able to free up sufficient memory to continue or it must quit.

State Transition Operation Triplet Errors

Many operations normally cause state transitions. If one of these operations fails, for example, returns TWRC_FAILURE, do not make the state transition. The application must check the Return Code following every operation and update the current state only if the operation succeeds.

An implied state transition during `DG_CONTROL/DAT_PENDINGXFERS/ MSG_ENDXFER` deserves special note here. If the `Count` field of the `TW_PENDINGXFERS` structure is zero then the source will automatically transition back to State 5. Application writers should be aware of this condition and react accordingly.

Error Handling and State Transitions

It is possible that during execution of any triplet that the data source will fail unexpectedly. It is very important that applications pay attention to the TWAIN State of the data source at the time of failure. A hanging or deadlock condition will occur if the application fails to recover from error conditions with the proper state transitions. Most error handling is fairly obvious, however the following items have been mishandled in the past.

Failing Transition to State 5

A data source may fail a call to DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS unexpectedly. It is important to note that if an application requests the User Interface be suppressed, and the data source returns a code of TWRC_CHECKSTATUS, this means only that User Interface suppression was not possible. The transition to State 5 still occurred. If the application does not like this condition, then it may call MSG_DISABLEDS to close the data source without further user interaction. A return code of TWRC_FAILURE indicates that the transition to State 5 has not occurred.

Failure During State 6 or 7

It is important to be aware that when an error occurs during image transfer, a state transition to State 5 is not implicit. A call to DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET or MSG_ENDXFER is required for a state transition back to State 5. If an applications calls MSG_DISABLEDS immediately after such a failure without first making the required calls to DAT_PENDINGXFERS, the resulting behavior of the data source will not be predictable. The data source should fail any call to MSG_DISABLEDS outside of State 5.

Requirements for an Application to be TWAIN-Compliant

Applications are required to support only a subset of the defined TWAIN operations. As an application advances its need to set attributes it will also need to implement a more complete set of the defined operations. This includes provision of support for more transfer mechanisms.

An application **must** support the following to be considered TWAIN-compliant:

Operations

The following six operations are consumed by the Source Manager:

DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS

DG_CONTROL / DAT_IDENTITY / MSG_OPENDS

DG_CONTROL / DAT_PARENT / MSG_CLOSEDSM

DG_CONTROL / DAT_PARENT / MSG_OPENDSM

DG_CONTROL / DAT_STATUS / MSG_GET

The following seven operations are consumed by a Source:

DG_CONTROL / DAT_CAPABILITY / MSG_GET

DG_CONTROL / DAT_CAPABILITY / MSG_SET

DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT

DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET

DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER

DG_CONTROL / DAT_STATUS / MSG_GET

DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS

DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

Notices

Every application must support receipt of two notices from Sources. These are:

MSG_XFERREADY	indicates application can initiate transfer
MSG_CLOSEDSREQ	indicates the Source needs to be disabled

Capabilities

Applications must support one capability:

CAP_XFERCOUNT	Application sets the maximum number of transfers a Source is allowed to offer per session.

Applications that consume image information should support negotiation with the following capabilities:

ICAP_XFERMECH	the transfer mechanism to be used for the next transfer
ICAP_UNITS	unit of measure for all measured values (default is inches)
ICAP_PIXELTYPE	how image data is interpreted (Color, Gray, B&W, etc.)

Source requirements for TWAIN-compliance are presented in Chapter 5.

4

Advanced Application Implementation

Using TWAIN to acquire a raster image from a device is relatively simple to implement as demonstrated in Chapter 3. However, TWAIN also allows application developers to go beyond the simple acquisition of a single image in Native (DIB or PICT) format. These more advanced topics are discussed in this chapter. They include:

Chapter Contents

Capabilities

Capabilities, and the power of an application to negotiate capabilities with the Source, give control to TWAIN-compliant applications. In Chapter 3, you saw the negotiation of one capability, CAP_XFERCOUNT. This capability was negotiated during State 4 as is always the case unless delayed negotiation is agreed to by both the application and Source. In fact, there is much more to know about capabilities.

Capability Values

Several values are used to define each capability. As seen in Chapter 9, TWAIN defines a Default Value and a set of Allowed Values for each of the capabilities. The application is not able to modify the Default Value. However, it is able to limit the values offered to a user to a subset of the Allowed Values and to select the capability's Current Value.

Default Value

When a Source is opened, the Current Values for each of its capabilities are set to the TWAIN Default Values listed in Chapter 9. If no default is defined by TWAIN, the Source will select a value for its default. An application can return a capability to its TWAIN-defined default by issuing a DG_CONTROL / DAT_CAPABILITY / MSG_RESET operation.

Although TWAIN defines defaults for many of the capabilities, a Source may have a different value that it would prefer to use as its default because it would be more efficient. For example, the Source may normally use a 0 bit in a black and white image to indicate white. However, the default for ICAP_PIXELFLAVOR is TWPF_CHOCOLATE which states that a 0 represents black. Although the TWAIN default is TWPF_CHOCOLATE, the Source's preferred default would be TWPF_VANILLA. When the application issues a DG_CONTROL / DAT_CAPABILITY / MSG_GETDEFAULT operation, the Source returns information about its preferred defaults. The Source and application may be able to negotiate a more efficient transfer based on this information.

Note that this does not imply that the TWAIN defaults should be completely disregarded. When trying to resolve the conflict between the "preferred" value of a particular data source capability and the TWAIN-specified default, it should be considered that the problem is similar to storing and restoring image attributes from session to session. It is reasonable to assume that a data source will want to store the current values for some capabilities to be restored as the current values in a future session. It is then also reasonable to expect that these restored values will be reflected as the current settings for the appropriate capabilities. While storing settings is only really useful for image attributes (the data source would not store the value of ICAP_PIXELFLAVOR, but it might store the current ICAP_RESOLUTION), it should be stated that preferred values of a data source are to be treated in the same manner.

At the time of loading the data source, all current values for the appropriate capabilities would be set to values that have either been restored from a previous session, or those that are "preferred" by the data source. This current value will remain until it has been explicitly changed by the calling application, or that application issues a MSG_RESET.

These are best illustrated using examples, since not all capabilities are suitable for preferred values, and most are not suitable to be stored and restored across multiple scanning sessions.

Example 1:
Scan Parameters are stored in one session and restored in another

1. User configures the data source User Interface with the following parameters: 4x6 inch image in 24-bit at 200 DPI X and Y resolution
2. User selects "Scan" and data source signals application to transfer.
3. Application acquires the image successfully.
4. Application disables the data source.
5. Application inquires during State 4 the current values of Frame, Pixel Type, Bit Depth, and Resolution.

6. Data source reports to each inquiry the current values that were set by the user: 4x6 inch image in 24-bit at 200 DPI X and Y resolution.
7. Application closes the data source.
8. During close procedure, the data source stores the current Frame, Pixel Type, Bit Depth and Resolution.
9. Application opens data source.
10. During open procedure, the data source restores current Frame, Pixel Type, Bit Depth and Resolution.
11. Application inquires during State 4 the current values of Frame, Pixel Type, Bit Depth, and Resolution.
12. Data source reports to each inquiry the current values that were restored from previous session: 4x6 inch image in 24-bit at 200 DPI X and Y resolution in one session.

Example 2: Data Source represents the preferred Pixel Flavor without compromising TWAIN Defined Default value

1. Application opens data source for the first time
2. Application inquires during State 4 about the Default Pixel Flavor
3. Data source reports that the default pixel flavor is TWPF_CHOCOLATE (see Chapter 9).
4. Application inquires during State 4 about the current pixel flavor.
5. Data source reports that the current pixel flavor is TWPF_VANILLA (because this device returns data in that gender natively).
6. Application issues reset to current pixel flavor.
7. During reset operation, data source changes current value to TWPF_CHOCOLATE and prepares to invert data during transfer to accommodate the calling application request.

There is a condition where this logic falls apart. If the data source wants to return a TW_ENUMERATION to a MSG_GET request for a constrained capability, there is a chance that the Default value imposed by the TWAIN Specification (Chapter 9) will not exist within the constrained set of values. In this case, the application should consider the default value to be undefined. Common sense should dictate that the data source provide some default that is reasonable within the currently available set of values for safety (a bad index in a TW_ENUMERATION could be a disaster). When the default value is actually used (during MSG_RESET) the constraints shall be lifted, and the original default value will once again exist and be defined. (See next section on Constrained Capabilities about MSG_RESET) This is only a problem with a TW_ENUMERATION container, since it contains an index to the default.

Current Value

The application may request to set the Current Value of a capability. If the Source's user interface is displayed, the Current Value should be reflected (perhaps by highlighting). If the application sets the Current Value, it will be used for the acquire and transfer unless the user or an automatic Source process changes it. The application can determine if changes were made by checking the Current Value during State 6.

To determine just the capability's Current Value, use DG_CONTROL / DAT_CAPABILITY / MSG_GETCURRENT. To determine both the Current Value and the Available Values, use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation. For example, you could do a MSG_GET on ICAP_PIXELTYPE and the Source might return a TW_ENUMERATION container containing TWPT_BW, TWPT_GRAY, and TWPT_RGB as Available Values.

To set the Current Value:

Use DG_CONTROL / DAT_CAPABILITY / MSG_SET and one of the following containers:

- **TWON_ONEVALUE:** Place the desired value in TW_ONEVALUE.Item.
- **TWON_ARRAY:** Place only the desired items in TW_ARRAY.ItemList.

These must be a subset of the items returned by the Source from a MSG_GET operation.

It is also possible to set Current Values using the TW_ENUMERATION and TW_RANGE containers. See the Available Values information for details.

Available Values

To limit the settings the Source can use during the acquire and transfer process, the application may be able to restrict the Available Values. The Source should not use a value outside these values. These restrictions should be reflected in the Source's user interface so unavailable values are not offered to the user.

For example, if the MSG_GET operation on ICAP_PIXELTYPE indicates the Source supports TWPT_BW, TWPT_GRAY, and TWPT_RGB images and the application only wants black and white images, it can request to limit the Available Values to black and white.

To limit the Available Values:

Use DG_CONTROL / DAT_CAPABILITY / MSG_SET and one of the following containers:

- **TWON_ENUMERATION:** Place only the desired values in the TW_ENUMERATION.ItemList field. The Current Value can also be set at this time by setting the CurrentIndex to point to the desired value in the ItemList.
- **TWON_RANGE:** Place only the desired values in the TW_RANGE fields. The current value can also be set by setting the CurrentValue field.

Note that TW_ONEVALUE and TW_ARRAY containers cannot be used to limit the Available Values.

Capability Negotiation

The negotiation process consists of three basic parts:

1. The application determines which capabilities a Source supports
2. The application sets the supported capabilities as desired
3. The application verifies that the settings were accepted by the Source

Negotiation (Part 1)
Application Determines Which Capabilities the Source Supports

Step 1

Application allocates a TW_CAPABILITY structure and fills its fields as follows:

- Cap = the CAP_ or ICAP_ name for the capability it is interested in
- ConType = TWON_DONTCARE16
- hContainer = NULL

Step 2

Application uses the TW_CAPABILITY structure in a DG_CONTROL / DAT_CAPABILITY / MSG_GET operation.

Step 3

The Source examines the Cap field to see if it supports the capability. If it does, it creates information for the application. In either case, it sets its Return Code appropriately.

Step 4

Application examines the Return Code, and maybe the Condition Code, from the operation.

If TWRC_SUCCESS then the Source does support the capability and

- The ConType field was filled by the Source with a container identifier (TWON_ARRAY, TWON_ENUMERATION, TWON_ONEVALUE, or TWON_RANGE)
- The Source allocated a container structure of ConType and referenced the hContainer field to this structure. It then filled the container with values describing the capability's Current Value, Default Value, and Available Values.

Based on the type of container and its contents (whose type is indicated by it ItemType field), the application can read the values. The application must deallocate the container.

If TWRC_FAILURE and TWCC_CAPUNSUPPORTED

- Source does not support this capability

The application can repeat this process for every capability it wants to learn about. If the application really only wants to get the Current Value for a capability, it can use the MSG_GETCURRENT operation instead. In that case, the ConType will just be TWON_ONEVALUE or TWON_ARRAY but not TWON_RANGE or TWON_ENUMERATION.

Note: The capability, CAP_SUPPORTEDCAPS, returns a list of capabilities that a Source supports. But it doesn't indicate whether the supported capabilities can be negotiated, If the Source does not support the CAP_SUPPORTEDCAPS capabilities, it returns TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Negotiation (Part 2)
The Application Sets the Supported Capability as Desired

Step 1

Application allocates a TW_CAPABILITY structure and fills its fields as follows:

- Cap = the CAP_ , ICAP_, or ACAP_name for the capability it is interested in
- ConType = TWON_ARRAY, TWON_ENUMERATION, TWON_ONEVALUE or TWON_RANGE (Refer to Chapter 9 to see each capability and what type(s) of container may be used to set a particular capability.)
- hContainer = The application must allocate a structure of type ConType and reference this field to it. (See the next step.)

Step 2

Application allocates a structure of type ConType and fills it. Based on values received from the Source during the MSG_GET, it can specify the desired Current Value and Available Values that it wants the Source to use. The application should not attempt to set the Source's Default Value, just put an appropriate constant in that field (ex. TWON_DONTCARE32).

Note: The application is responsible for deallocating the container structure when the operation is finished.

Step 3

Send the request to the Source using DG_CONTROL / DAT_CAPABILITY / MSG_SET.

Negotiation (Part 3)
The Application MUST Verify the Result of Their Request

Step 1

Even if a Source supports a particular capability, it is not required to support the setting of that capability. The application must examine the Return Code from the MSG_SET request to see what took place.

If TWRC_SUCCESS then the Source set the capability as requested.

If TWRC_CHECKSTATUS then

- The Source could not use one or more of your exact values. For instance, you asked for a value of 310 but it could only accept 100, 200, 300, or 400. Your request was within its legitimate range so it rounded it to its closest valid setting.

Use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation to determine the current and available settings at this time. This is the only way to determine if the Source's choice was acceptable to your application.

If TWRC_FAILURE / TWCC_BADVALUE then

- Either the Source is not granting your request to set or restrict the value.
- Or, your requested values were not within its range of legitimate values. It may have attempted to set the value to its closest available value.

Use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation to determine the current and available settings at this time. This is the only way to determine if your application can continue without your requested values.

You can repeat the setting and verifying processes for every capability of interest to your application. Remember, your application must deallocate all container structures.

The Most Common Capabilities

TWAIN defines over fifty capabilities. Although the number may seem overwhelming, it is easier to handle if you recognize that some of the capabilities are more commonly used. Here are some of these capabilities:

Basic Capabilities

Units

The ICAP_UNITS capability determines the unit of measure which will be used by the Source. The default is inches but centimeters, pixels, etc. are allowed. This capability's value is used when measuring several other values in capabilities and data structures including:

ICAP_PHYSICALHEIGHT,

ICAP_PHYSICALWIDTH,

ICAP_XNATIVERESOLUTION,

ICAP_YNATIVERESOLUTION,

ICAP_XRESOLUTION,

ICAP_YRESOLUTION,

TW_FRAME,

TW_IMAGEINFO.XResolution,

TW_IMAGEINFO.YResolution

Sense of the Pixel

The ICAP_PIXELFLAVOR specifies how a bit of data should be interpreted when transferred from Source to application. The default is TWPF_CHOCOLATE which means a 0 indicates black (or the darkest color). The alternative, TWPF_VANILLA, means a 0 indicates white (or the lightest color).

Resolution

The image resolution is reported in the TW_IMAGEINFO structure. To inquire or set the Source's resolution, use ICAP_XRESOLUTION and ICAP_YRESOLUTION.

Refer also to ICAP_XNATIVERESOLUTION and ICAP_YNATIVERESOLUTION.

Image Type Capabilities

Types of Pixel

The application should negotiate ICAP_PIXELTYPE and ICAP_BITDEPTH unless it can handle all pixel types at all bit depths. The allowed pixel types are: TWPT_BW, TWPT_GRAY, TWPT_RGB, TWPT_PALETTE, TWPT_CMY, TWPT_CMYK, TWPT_YUV, TWPT_YUVK, and TWPT_CIEXYZ.

Depth of the Pixels (in bits)

A pixel type such as TWPT_BW allows only 1 bit per pixel (either black or white). The other pixel types may allow a variety of bits per pixel (4-bit or 8-bit gray, 8-bit or 24-bit color). Be sure to set the ICAP_PIXELTYPE first, then set the ICAP_BITDEPTH.

Parameters for Acquiring the Image

Exposure

Several capabilities can influence this. They include ICAP_BRIGHTNESS, ICAP_CONTRAST, ICAP_SHADOW, ICAP_HIGHLIGHT, ICAP_GAMMA, and ICAP_AUTOBRIGHT.

Scaling

To instruct a Source to scale an image before transfer, refer to ICAP_XSCALING and ICAP_YSCALING.

Rotation

To instruct a Source to rotate the image before transfer, refer to ICAP_ROTATION and ICAP_ORIENTATION.

Constrained Capabilities and Message Responses

There is some confusion about how the data source should respond to various capability queries when the application has imposed constraints upon the supported values. The following guidelines should help clarify the situation.

MSG_RESET

It is known that this call resets the current value of the requested capability to the default. It must also be stated that this call will also reset any application imposed constraints upon the requested capability.

MSG_GETCURRENT, and MSG_GETDEFAULT

It is intuitive to assume that this message should not be supported by capabilities that have no Current or Default value. However, the specification says otherwise in Chapter 9 (a good example is **ICAP_SUPPORTEDCAPS**). In this case, it makes sense to simply respond to these messages in the same manner as **MSG_GET**.

It can also be assumed that it is more intuitive for a data source to respond to this capability with a **TW_ONEVALUE** container in all cases that a **TW_ONEVALUE** container is allowed.

MSG_GET

If an application has constrained the current capability, then the data source response to this message should reflect those constraints. Otherwise, this should respond with all the values that the data source supports. Of course, the number of values that can be placed in the response are restricted by the allowed containers for the particular current capability outlined in Chapter 9.

MSG_SET

As indicated in the Chapter 7 description of this capability triplet:

> *"Current Values are set when the container is a TW_ONEVALUE or TW_ARRAY. Available and Current Values are set when the container is a TW_ENUMERATION or TW_RANGE."*

To further clarify this operation, it should be stated that when an application imposes a constraint, the data source must consider the set of supported values and the set of requested constraints. The resulting set of values shall contain only the values that are shared by those supported and those requested.

A condition may arise after constraints are imposed, where the default value is no longer within the set of supported values. When using a TW_ENUMERATION, the reported default index should be changed by the data source to something that falls within the new constrained set. This is simply a precaution to ensure it is a valid index. In this case, the Default index in a TW_ENUMERATION loses meaning and should be ignored by applications, since MSG_RESET shall cause the constraints to be eliminated.

Capability Containers in Code Form

Capability information is passed between application and Source by using data structures called containers: TW_ARRAY, TW_ENUMERATION, TW_ONEVALUE, and TW_RANGE. The actions needed to create (pack) and read (unpack) containers are illustrated here in the following code segments. Containers are flexible in that they can be defined to contain one of many types of data. Only one ItemType (TWTY_xxxx) is illustrated per Container (TWON_xxxx) here. Refer to the toolkit disk for complete packing and unpacking utilities that you can use with containers.

Reading (unpacking) a Container from a MSG_GET Operation

```
//-----------------------------------------------------
//Example of DG_CONTROL / DAT_CAPABILITY / MSG_GET
//-----------------------------------------------------
TW_CAPABILITY   twCapability;
TW_INT16         rc;

//Setup TW_CAPABILITY Structure
   twCapability.Cap = Cap;     //Fill in capability of interest
   twCapability.ConType = TWON_DONTCARE16;
   twCapability.hContainer = NULL;
```

```
//Send the Triplet
   rc = (*pDSM_Entry)(&AppID,
                &SourceID,
                DG_CONTROL,
                DAT_CAPABILITY,
                MSG_GET,
               (TW_MEMREF)&twCapability);
//Check return code
   if (rc == TWRC_SUCCESS)
{
//Switch on Container Type
       switch (twCapability.ConType)
       {
//-----ENUMERATION
          case TWON_ENUMERATION:
          {
          pTW_ENUMERATION   pvalEnum;
          TW_UINT16         valueU16;
          TW_UINT16         index;
          pvalEnum =
(pTW_ENUMERATION)GlobalLock(twCapability.hContainer);
          NumItems = pvalEnum->NumItems;
          CurrentIndex = pvalEnum->CurrentIndex;
          DefaultIndex = pvalEnum->DefaultIndex;
          for (index = 0; index < pvalEnum->NumItems; index++)
          {
              if (pvalEnum->ItemType == TWTY_UINT16)
              {
                  valueU16 = ((TW_UINT16)(pvalEnum->ItemList[index*2]));
                  //Store Item Value
              }
          }
          GlobalUnlock(twCapability.hContainer);
          }
          break;
//-----ONEVALUE
          case TWON_ONEVALUE:
          {
          pTW_ONEVALUE      pvalOneValue;
          TW_BOOL         valueBool;
         pvalOneValue =
(pTW_ONEVALUE)GlobalLock(twCapability.hContainer);
         if (pvalOneValue->ItemType == TWTY_BOOL)
         {
               valueBool = (TW_BOOL)pvalOneValue->Item;
               //Store Item Value
         }
         GlobalUnlock(twCapability.hContainer);
         }
         break;
//-----RANGE
          case TWON_RANGE:
          {
          pTW_RANGE         pvalRange;
          pTW_FIX32         pTWFix32;
          float            valueF32;
          TW_UINT16         index;
            pvalRange = (pTW_RANGE)GlobalLock(twCapability.hContainer);
            if ((TW_UINT16)pvalRange->ItemType == TWTY_FIX32)
            {
               pTWFix32 = &(pvalRange->MinValue);
               valueF32 = FIX32ToFloat(*pTWFix32);
               //Store Item Value
               pTWFix32 = &(pvalRange->MaxValue);
               valueF32 = FIX32ToFloat(*pTWFix32);
               //Store Item Value
               pTWFix32 = &(pvalRange->StepSize);
               valueF32 = FIX32ToFloat(*pTWFix32);
               //Store Item Value
            }
            GlobalUnlock(twCapability.hContainer);
         }
         break;
//-----ARRAY
          case TWON_ARRAY:
          {
          pTW_ARRAY         pvalArray;
          TW_UINT16         valueU16;
          TW_UINT16         index;
            pvalArray = (pTW_ARRAY)GlobalLock(twCapability.hContainer);
            for (index = 0; index < pvalArray->NumItems; index++)
            {
               if (pvalArray->ItemType == TWTY_UINT16)
               {
                  valueU16 = ((TW_UINT16)(pvalArray->ItemList[index*2]));
                  //Store Item Value
               }
            }
            GlobalUnlock(twCapability.hContainer);
         }
         break;
      }   //End Switch Statement
     GlobalFree(twCapability.hContainer);
   } else {
      //Capability MSG_GET Failed check Condition Code
   }
```

```
/**********************************************************
* Fix32ToFloat
* Convert a FIX32 value into a floating point value.
**********************************************************/
float FIX32ToFloat (TW_FIX32    fix32)
{
   float    floater;

   floater = (float)fix32.Whole + (float)fix32.Frac / 65536.0;
   return floater;
}
```

Creating (packing) a Container for a MSG_SET Operation

```
//-----------------------------------------------
//Example of DG_CONTROL / DAT_CAPABILITY / MSG_SET
//-----------------------------------------------
TW_CAPABILITY   twCapability;
TW_INT16         rc;
TW_UINT32         NumberOfItems;

   twCapability.Cap = Cap;      //Insert Capability of Interest
   twCapability.ConType = Container;
      //Use TWON_ONEVALUE or TWON_ARRAY to set current value
      //Use TWON_ENUMERATION or TWON_RANGE to limit available values

   switch (twCapability.ConType)
   {
//-----ENUMERATION
      case TWON_ENUMERATION:
      {
      pTW_ENUMERATION   pvalEnum;

         //The number of Items in the ItemList
         NumberOfItems = 2;


       //Allocate memory for the container and additional ItemList
       // entries
       twCapability.hContainer = GlobalAlloc(GHND,
          (sizeof(TW_ENUMERATION) + sizeof(TW_UINT16) *
(NumberOfItems)));
       pvalEnum = (pTW_ENUMERATION)GlobalLock(twCapability.hContainer);

       pvalEnum->NumItems = 2      //Number of Items in ItemList
       pvalEnum->ItemType = TWTY_UINT16;
       ((TW_UINT16)(pvalEnum->ItemList[0])) = 1;
       ((TW_UINT16)(pvalEnum->ItemList[1])) = 2;

       GlobalUnlock(twCapability.hContainer);
    }
    break;
//-----ONEVALUE
      case TWON_ONEVALUE:
      {
      pTW_ONEVALUE      pvalOneValue;

         twCapability.hContainer = GlobalAlloc(GHND,
sizeof(TW_ONEVALUE));
         pvalOneValue =
(pTW_ONEVALUE)GlobalLock(twCapability.hContainer);

         (TW_UINT16)pvalOneValue->ItemType = TWTY_UINT16;
         (TW_UINT16)pvalOneValue->Item = 1;

         GlobalUnlock(twCapability.hContainer);
      }
      break;

//-----RANGE
      case TWON_RANGE:
      {
      pTW_RANGE         pvalRange;
      TW_FIX32         TWFix32;
      float            valueF32;

         twCapability.hContainer = GlobalAlloc(GHND, sizeof(TW_RANGE));
         pvalRange = (pTW_RANGE)GlobalLock(twCapability.hContainer);

         (TW_UINT16)pvalRange->ItemType = TWTY_FIX32;
         valueF32 = 100;
         TWFix32 = FloatToFIX32 (valueF32);
         pvalRange->MinValue = *((pTW_INT32) &TWFix32);
         valueF32 = 200;
         TWFix32 = FloatToFIX32 (valueF32);
         pvalRange->MaxValue = *((pTW_INT32) &TWFix32);

         GlobalUnlock(twCapability.hContainer);
      }
      break;

//-----ARRAY
      case TWON_ARRAY:
      {
      pTW_ARRAY         pvalArray;

         //The number of Items in the ItemList
         NumberOfItems = 2;

       //Allocate memory for the container and additional ItemList
entries
       twCapability.hContainer = GlobalAlloc(GHND,
         (sizeof(TW_ARRAY) + sizeof(TW_UINT16) * (NumberOfItems)));
       pvalArray = (pTW_ARRAY)GlobalLock(twCapability.hContainer);

         (TW_UINT16)pvalArray->ItemType = TWTY_UINT16;
         (TW_UINT16)pvalArray->NumItems = 2;
         ((TW_UINT16)(pvalArray->ItemList[0])) = 1;
         ((TW_UINT16)(pvalArray->ItemList[1])) = 2;

         GlobalUnlock(twCapability.hContainer);
      }
      break;
   }
```

```
//-----MSG_SET
   rc = (*pDSM_Entry)(&AppID,
                  &SourceID,
                  DG_CONTROL,
                  DAT_CAPABILITY,
                  MSG_SET,
                  (TW_MEMREF)&twCapability);
   GlobalFree(twCapability.hContainer);
   switch (rc)
   {
      case TWRC_SUCCESS:
         //Capability's Current or Available value was set as specified
      case TWRC_CHECKSTATUS:
        //The Source matched the specified value(s) as closely as
possible
         //Do a MSG_GET to determine the settings made
      case TWRC_FAILURE:
         //Check the Condition Code for more information
   }
/***********************************************************
* FloatToFix32
* Convert a floating point value into a FIX32.
***********************************************************/
TW_FIX32 FloatToFix32 (float floater)
{
   TW_FIX32 Fix32_value;
   TW_INT32 value = (TW_INT32) (floater * 65536.0 + 0.5);
   Fix32_value.Whole = value >> 16;
   Fix32_value.Frac = value & 0x0000ffffL;
   return (Fix32_value);
}
```

Delayed Negotiation - Negotiating Capabilities After State 4

Applications may inquire about a Source’s capability values at any time during the session with the Source. However, as a rule, applications can only request to **set** a capability during State 4. The rationale behind this restriction is tied to the display of the Source’s user interface when the Source is enabled. Many Sources will modify the contents of their user interface in response to some of the application’s requested settings. These user interface modifications prevent the user from selecting choices that do not meet the application’s requested values. The Source’s user interface is never displayed in State 4 so changes can be made without the user’s awareness. However, the interface may be displayed in States 5 through 7.

Some capabilities have no impact on the Source’s user interface and the application may really want to set them later than State 4. To allow delayed negotiation, the application must request, during State 4, that a particular capability be able to be set later (during States 5 or 6). The Source may agree to this request or deny it. The request is negotiated by the application with the Source by using the DG_CONTROL / DAT_CAPABILITY operations on the CAP_EXTENDEDCAPS capability.

On the CAP_EXTENDEDCAPS capability, the DG_CONTROL / DAT_CAPABILITY operations:

MSG_GET

Indicates all capabilities that the Source is willing to negotiate in State 5 or 6.

MSG_SET

Specifies which capabilities the application wishes to negotiate in States 5 or 6.

MSG_GETCURRENT

Provides a list of all capabilities which the Source and application have agreed to allow to be negotiated in States 5 or 6.

As with any other capability, if the Source does not support negotiating CAP_EXTENDEDCAPS, it will return the Return Code TWRC_FAILURE with the Condition Code TWCC_CAPUNSUPPORTED.

If an application attempts to set a capability in State 5 or 6 and the Source has not previously agreed to this arrangement, the operation will fail with a Return Code of TWRC_FAILURE and a Condition Code of TWCC_SEQERROR.

If an application does not use the Source's user interface but presents its own, the application controls the state of the Source explicitly. If the application wants to set the value of any capability, it returns the Source to State 4 and does so. Therefore, an application using its own user interface will probably not need to use CAP_EXTENDEDCAPS.

Options for Transferring Data

As discussed previously, there are three modes defined by TWAIN for transferring data:

- Native
- Disk File
- Buffered Memory

A Source is required to support Native and Buffered Memory transfers.

Native Mode Transfer

The use of Native mode, the default mode, for transferring data was covered in Chapter 3. There is one potential limitation that can occur in a Native mode transfer. That is, there may not be an adequately large block of RAM available to hold the image. This situation will not be discovered until the transfer is attempted when the application issues the DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET operation.

When the lack of memory appears, the Source may respond in one of several ways. It can:

- Simply fail the operation.
- Clip the image to make it fit in the available RAM - The Source should notify the user that the clipping operation is taking place due to limited RAM. The clipping should maintain both the aspect ratio of the selected image and the origin (upper-left).
- Interact with the user to allow them to resize the image or cancel the capture.

The Return Code / Condition Code returned from the DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET operation may indicate one of these actions occurred.

If the Return Code is TWRC_XFERDONE:

This indicates the transfer was completed and the session is in State 7. However, it does not guarantee that the Source did not clip the image to make it fit. Even if the application issued a DG_IMAGE / DAT_IMAGEINFO / MSG_GET operation prior to the transfer to determine the image size, it cannot assume that the ImageWidth and ImageLength values returned from that operation really apply to the image that was ultimately transferred. If the dimensions of the image are important to the application, the application should always check the actual transferred image size after the transfer is completed. To do this:

1. Execute a DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation to move the session from State 7 to State 6 (or 5).
2. Determine the actual size of the image that was transferred:
   a. **On Windows -** Read the DIB header
   b. **On Macintosh -** Check the picFrame in the Picture

If the Return Code is TWRC_CANCEL:

The acquisition was canceled by the user. The session is in State 7. Execute a DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation to move the session from State 7 to State 6 (or 5).

If the Return Code is TWRC_FAILURE:

Check the Condition Code to determine the cause of the failure. The session is in State 6. No memory was allocated for the DIB or PICT. The image is still pending. If lack of memory was the cause, you can try to free additional memory or discard the pending image by executing DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER.

Disk File Mode Transfer

Determine if a Source Supports the Disk File Mode

- Use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation.
- Set the TW_CAPABILITY's Cap field to ICAP_XFERMECH.
- The Source returns information about the transfer modes it supports in the container structure pointed to by the hContainer field of the TW_CAPABILITY structure. The disk file mode is identified as TWSX_FILE. Sources are not required to support Disk File Transfer so it is important to verify its support.

After Verifying Disk File Transfer is Supported, Set Up the Transfer

During State 4:

- Set the ICAP_XFERMECH to TWSX_FILE. Use the DG_CONTROL / DAT_CAPABILITY / MSG_SET operation.
- Use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation to determine which file formats the Source can support. Set TW_CAPABILITY.Cap to ICAP_IMAGEFILEFORMAT and execute the MSG_GET. The Source returns the supported format identifiers which start with TWFF_ and may include TWFF_PICT, TWFF_BMP, TWFF_TIFF, etc. They are listed in the TWAIN.H file and in the Constants section of Chapter 8.

During States 4, 5, or 6:

To set up the transfer, the DG_CONTROL / DAT_SETUPFILEXFER operations of MSG_GET, MSG_GETDEFAULT, and MSG_SET can be used. The data structure used in the DSM_Entry call is a TW_SETUPFILEXFER structure:

```
typedef struct {
  TW_STR255   FileName; /* File to contain data       */
  TW_UINT16   Format;   /* A TWFF_xxxx constant        */
  TW_HANDLE   VRefNum;  /* Used for Macintosh only    */
} TW_SETUPFILEXFER, FAR *pTW_SETUPFILEXFER;
```

The application could use the MSG_GETDEFAULT operation to determine the default file format and filename (TWAIN.TMP in the current directory). If acceptable, the application could just use that file. However, most applications prefer to set their own values for filename and format. The MSG_SET operation allows this. It is done during State 6. To set your own filename and format, do the following:

1. Create the file specified for the transfer and close it. Be sure the Source has permission to read and write this file.
2. Allocate the required TW_SETUPFILEXFER structure. Then, fill in these fields:
   a. **FileName -** the desired file name. On Windows, be sure to include the complete path name.
   b. **Format -** the constant for the desired, and supported, format. (TWFF_xxxx). If you set it to an unsupported format, the operation returns TWRC_FAILURE / TWCC_BADVALUE and the Source resets itself to write data to its default file.
   c. **VRefNum -** On Macintosh, write the file's volume reference number. On Windows, fill the field with a TWON_DONTCARE16.
3. Invoke the DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation.

Execute the Transfer into the File

After the application receives the MSG_XFERREADY notice from the Source and has issued the DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET operation:

Use the following operation: DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET

This operation does not have an associated data structure but just uses NULL for the pData parameter in the DSM_Entry call.

- If the application has not specified a filename (during the setup) - the Source will use either its default file or the last file information it was given.
- If the file specified by the application does not exist - the Source should create it.
- If the file exists but already has data in it - the Source should overwrite the existing data. Notice, if you are transferring multiple files and using the same file name each time, you will overwrite the data unless you copy it to a different filename between transfers.

Note: The application cannot abort a Disk File transfer once initiated. However, the Source's user interface may allow the user to cancel the transfer.

Following execution, be sure to check the Return Code:

TWRC_XFERDONE: File was written successfully. The application needs to invoke the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER to transition the session back to State 6 (or 5) as was illustrated in Chapter 3.

TWRC_CANCEL: The user canceled the transfer. The contents of the file are undefined. Invoke DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER to transition the session back to State 6 (or 5) as was illustrated in Chapter 3.

TWRC_FAILURE
The Source remained in State 6.
The contents of the file are undefined.
The image is still pending. To discard it, use DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER.

Check the Condition Code to determine the cause of the failures. The alternatives are:

TWCC_BADDEST = Operation aimed at invalid Source

TWCC_OPERATIONERROR = Either the file existed but could not be accessed or a system error occurred during the writing

TWCC_SEQERROR = Operation invoked in invalid state (i.e. not 6)

Buffered Memory Mode Transfer

Set Capability Values for the Buffered Memory Mode, if Desired

Data is typically transferred in uncompressed format. However, if you are interested in knowing if the Source can transfer compressed data when using the buffered memory mode, perform a DG_CONTROL / DAT_CAPABILITY / MSG_GET on the ICAP_COMPRESSION. The values will include TWCP_NONE (the default) and perhaps others such as TWCP_PACKBITS, TWCP_JPEG ,etc. (See the list in the Constants section of Chapter 8.) More information on compression is available later in this chapter in the section called Transfer of Compressed Data.

Set up the Transfer

During State 4:

Set the ICAP_XFERMECH to TWSX_MEMORY by using the DG_CONTROL / DAT_CAPABILITY / MSG_SET operation.

During States 4, 5, or 6:

The DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET operation is used by the application to determine what buffer sizes the Source wants to use during the transfer. The Source might have more accurate information in State 6.

The data structure used in the DSM_Entry call is a TW_SETUPMEMXFER structure:

```
typedef struct {
  TW_UINT32   MinBufSize  /* Minimum buffer size in bytes   */
  TW_UINT32   MaxBufSize  /* Maximum buffer size in bytes   */
  TW_UINT32   Preferred   /* Preferred buffer size in bytes */
} TW_SETUPMEMXFER, FAR *pTW_SETUPMEMXFER;
```

The Source will fill in the appropriate values for its device.

Buffers Used for Uncompressed Strip Transfers

- The application is responsible for allocating and deallocating all memory used during the buffered memory transfer.
- For optimal performance, create buffers of the Preferred size.
- In all cases, the size of the allocated buffers must be within the limits of MinBufSize to MaxBufSize. If outside of these limits, the Source will fail the transfer operation with a Return Code of TWRC_FAILURE / TWCC_BADVALUE.
- If using more than one buffer, all buffers must be the same size.
- Raster lines must be double-word aligned and padded with zeros is recommended .

Execute the Transfer Using Buffers

After the application receives the MSG_XFERREADY notice from the Source and has issued the DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET operation:

- Allocate one or more buffers of the same size. The best size is the one indicated by the TW_SETUPMEMXFER.Preferred field. If that is impossible, be certain the buffer size is between MinBufSize and MaxBufSize.
- Allocate the TW_IMAGEMEMXFER structure. It will be used in the DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET operation.

The TW_IMAGEMEMXFER structure looks like this:

```
typedef struct {
  TW_UINT16  Compression;
  TW_UINT32  BytesPerRow;
  TW_UINT32  Columns;
  TW_UINT32  Rows;
  TW_UINT32  XOffset;
  TW_UINT32  YOffset;
  TW_UINT32  BytesWritten;
  TW_MEMORY  Memory;
} TW_IMAGEMEMXFER, FAR *pTW_IMAGEMEMXFER;
```

Fill in the TW_IMAGEMEMXFER's first field with TWON_DONTCARE16 and the following six fields with TWON_DONTCARE32.

The TW_MEMORY structure embedded in there looks like this:

```
typedef struct {
  TW_UINT32  Flags;
  TW_UINT32  Length;
  TW_MEMREF  TheMem;
} TW_MEMORY, FAR *pTW_MEMORY;
```

Fill in the TW_MEMORY structure as follows:

Memory.Flags

Place TWMF_APPOWNS bit-wise ORed with TWMF_POINTER or TWMF_HANDLE

Memory.Length

The size of the buffer in bytes

Memory.TheMem

A handle or pointer to the memory buffer allocated above (depending on which one was specified in the Flags field).

Following each buffer transfer, the Source will have filled in all the fields except Memory which it uses as a reference to the memory block for the data.

The flow of the transfer of buffers is as follows:

Step 1

Buffered Memory transfers provide no embedded header information. Therefore, the application must determine the image attributes. After receiving the MSG_XFERREADY, i.e. while in State 6, the application issues the DG_IMAGE / DAT_IMAGEINFO / MSG_GET and DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET operations to learn about the image's bitmap characteristics and the size and location of the original image on the original page (before scaling or other processing). If additional information is desired, use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation.

Step 2

The application issues DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET.

Step 3

The application checks the Return Code.

If TWRC_SUCCESS:

Examine the TW_IMAGEMEMXFER structure for information about the buffer. If you plan to reuse the buffer, copy the data to another location.

Loop back to Step 2 to get another buffer. Be sure to reinitialize the information in the TW_IMAGEMEMXFER structure (including the Memory fields), if necessary. Issue another DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET operation.

If TWRC_XFERDONE:

This is how the Source indicates it just transferred the last buffer successfully. Examine the TW_IMAGEMEMXFER structure for information about the buffer. Perhaps, copy the data to another location, as desired, then go to Step 4.

If TWRC_CANCEL:

The user aborted the transfer. The application must send a DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER as described in Chapter 3 to move from State 7 to State 6 (or 5).

If TWRC_FAILURE:

Examine the Condition Code to determine the cause and handle it.
If the failure occurred during the transfer of the first buffer, the session is in State 6. If the failure occurred on a subsequent buffer, the session is in State 7.

The contents of the buffer are invalid and the transfer of the buffer is still pending. To abort it, use DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER.

Step 4

Once the TWRC_XFERDONE has been returned, the application must send the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER to conclude the transfer. This was described in Chapter 3 in the section called State 7 to 6 to 5 - Conclude the Transfer.

Note: The majority of Sources divide the image data into strips when using buffered transfers. A strip is a horizontal band starting at the leftmost side of the image and spanning the entire width but covering just a portion of the image length. The application can verify that strips are being used if the information returned from the Source in the TW_IMAGEMEMXFER structure’s XOffset field is zero and the Columns field is equal to the value in the TW_IMAGEINFO structure’s ImageWidth field.

An alternative to strips is the use of tiles although they are used by very few Sources. Refer to the TW_IMAGEMEMXFER information in Chapter 8 for an illustration of tiles.

The Image Data and Its Layout

The image which is transferred from the Source to the application has several attributes. Some attributes describe the size of the image. Some describe where the image was located on the original page. Still others might describe information such as resolution or number of bits per pixel. TWAIN provides means for the application to learn about these attributes.

Users are often able to select and modify an image's attributes through the Source's user interface. Additionally, TWAIN provides capabilities and operations that allow the application to impact these attributes prior to acquisition and transfer.

Getting Information About the Image That will be Transferred

Before the transfer occurs, while in State 6, the Source can provide information to the application about the actual image that it is about to transfer. Note, the information is lost once the transfer takes place so the application should save it, if needed. This information can be retrieved through two operations:

- DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
- DG_IMAGE / DAT_IMAGEINFO / MSG_GET

The area of an image to be acquired will always be a rectangle called a frame. There may be one or more frames located on a page. Frames can be selected by the user or designated by the application. The TW_IMAGELAYOUT structure communicates where the image was located on the original page relative to the origin of the page. It also indicates, in its FrameNumber field, if this is the first frame, or a later frame, to be acquired from the page.

The TW_IMAGELAYOUT structure looks like this:

```
typedef struct {
   TW_FRAME       Frame;
   TW_UINT32      DocumentNumber;
   TW_UINT32      PageNumber;
   TW_UINT32      FrameNumber;
   } TW_IMAGELAYOUT, FAR *pTW_IMAGELAYOUT;
```

The TW_FRAME structure specifies the values for the Left, Right, Top, and Bottom of the frame to be acquired. Values are given in ICAP_UNITS.

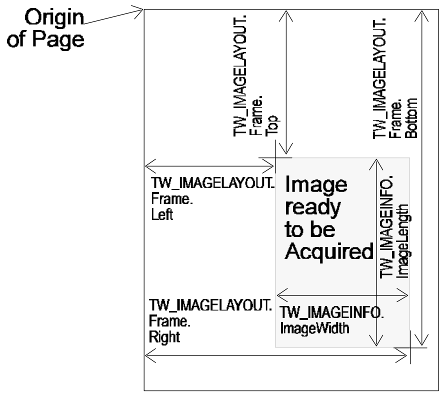

Origin of Page
TW_IMAGELAYOUT. Frame. Top
TW_IMAGELAYOUT. Frame. Bottom
TW_IMAGELAYOUT. Frame. Left
Image ready to be Acquired
TW_IMAGEINFO. ImageLength
TW_IMAGELAYOUT. Frame. Right
TW_IMAGEINFO. ImageWidth

Figure 4-1. TW_FRAME Structure

The DG_IMAGE / DAT_IMAGEINFO / MSG_GET operation communicates other attributes of the image being transferred. The TW_IMAGEINFO structure looks like this:

```
typedef struct {
   TW_FIX32    XResolution;
   TW_FIX32    YResolution;
   TW_INT32    ImageWidth;
   TW_INT32    ImageLength;
   TW_INT16    SamplesPerPixel;
   TW_INT16    BitsPerSample[8];
   TW_INT16    BitsPerPixel;
   TW_BOOL     Planar;
   TW_INT16    PixelType;
   TW_UINT16   Compression;
   } TW_IMAGEINFO, FAR * pTW_IMAGEINFO;
```

Notice how the ImageWidth and ImageLength relate to the frame described by the TW_IMAGELAYOUT structure.

Changing the Image Attributes

Normally, the user will select the desired attributes. However, the application may wish to do this initially during State 4. For example, if the user interface will not be displayed, the application may wish to select the frame. The application can use a DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET operation to define the area (frame) to be acquired. Although, there is no corresponding DG_IMAGE / DAT_IMAGEINFO / MSG_SET operation, the application can change those attributes by setting capabilities and the TW_IMAGELAYOUT data structure.

Here are the relationships:

TW_IMAGEINFO fields	**Capability or data structure that impacts the attribute**
XResolution	ICAP_XRESOLUTION
YResolution	ICAP_YRESOLUTION
ImageWidth	TW_IMAGELAYOUT.TW_FRAME.Right - TW_FRAME.Left **
ImageLength	TW_IMAGELAYOUT.TW_FRAME.Bottom - TW_FRAME.Top **
SamplesPerPixel	ICAP_PIXELTYPE (i.e. TWPT_BW has 1, TWPT_RGB has 3)
BitsPerSample	Calculated by BitsPerPixel divided by SamplesPerPixel
BitsPerPixel	ICAP_BITDEPTH
Planar	ICAP_PLANARCHUNKY
PixelType	ICAP_PIXELTYPE
Compression	ICAP_COMPRESSION

** ImageWidth and ImageLength are actually provided in pixels whereas TW_FRAME uses ICAP_UNITS.

Resolving Conflict Between ICAP_FRAMES, ICAP_SUPPORTEDSIZES, DAT_IMAGELAYOUT

Since there are several ways to negotiate the scan area, it becomes confusing when deciding what should take precedence. It is logical to assume that the last method used to set the frame will dictate the current frame. However, it may still be confusing to decide how that is represented during a MSG_GET operation for any of the three methods. The following behavior is suggested.

Note: Frame extents are only limited by ICAP_PHYSICALWIDTH and ICAP_PHYSICALHEIGHT. Setting ICAP_SUPPORTEDSIZES does NOT imply a new extent limitation. TWSS_xxxx sizes are simply predefined fixed frame sizes.

- **If the frame is set in DAT_IMAGELAYOUT**
  - ✓ ICAP_FRAMES shall respond to MSG_GETCURRENT with the dimensions of the frame set in the DAT_IMAGELAYOUT call.
  - ✓ ICAP_SUPPORTEDSIZES shall respond to MSG_GETCURRENT with TWSS_NONE
- **If the current frame is set from ICAP_FRAMES**
  - ✓ DAT_IMAGELAYOUT shall respond with the dimensions of the current frame set in ICAP_FRAMES
  - ✓ ICAP_SUPPORTEDSIZES shall respond to MSG_GETCURRENT with TWSS_NONE
- **If the current fixed frame is set from ICAP_SUPPORTEDSIZES**
  - ✓ DAT_IMAGELAYOUT shall respond to MSG_GET with the dimensions of the fixed frame specified in ICAP_SUPPORTEDSIZES
  - ✓ ICAP_FRAMES shall respond to MSG_GETCURRENT with the dimensions of the fixed frame specified in ICAP_SUPPORTEDSIZES

ICAP_ROTATION, ICAP_ORIENTATION Affect on ICAP_FRAMES, DAT_IMAGELAYOUT, DAT_IMAGEINFO

There is considerable confusion when trying to resolve the affect of Rotation and Orientation on the current frames and image layout. After careful consideration of the specification it has been concluded that ICAP_ROTATION and ICAP_ORIENTATION shall be applied after considering ICAP_FRAMES and DAT_IMAGELAYOUT.

Obviously a change in orientation will have an effect on the output image dimensions, so these must be reflected in DAT_IMAGEINFO during State 6. The resulting image dimensions shall be reported by the data source after considering the affect of the rotation on the current frame.

ICAP_ORIENTATION and ICAP_ROTATION are additive. The original frame is modified by ICAP_ORIENTATION as it is downloaded to the device by the Source, and represents the orientation of the paper being scanned. ICAP_ROTATION is then applied to the captured image to yield the final framing information that is reported to the Application in State 6 or 7. One possible reason for combining these two values is to use them to cancel each other out. For instance, some scanners with automatic document feeders may receive a performance benefit from describing an ICAP_ORIENTATION of TWOR_LANDSCAPE in combination with an ICAP_ROTATION of 90 degrees. This would allow the user to feed images in a landscape orientation (which lets them feed faster), while rotating the captured images back to portrait (which is the way the user wants to view them).

Transfer of Multiple Images

Chapter 3 discussed the transfer of a single image. Transferring multiple images simply requires looping through the single-image transfer process repeatedly whenever more images are available. Two classes of issues arise when considering multiple image transfer under TWAIN:

- What state transitions are allowable when a session is at an inter-image boundary?
- What facilities are available to support the operation of a document feeder? This includes issues related to high-performance scanning.

This section starts with a review of the single-image transfer process. This is followed by a discussion of options available to an application once the transfer of a single image is complete. Finally, document feeder issues are presented.

To briefly review the single-image transfer process:

- The application enables the Source and the session moves from State 4 to State 5.
- The Source sends the application a MSG_XFERREADY when an image is ready for transfer.
- The application uses DG_IMAGE / DAT_IMAGEINFO / MSG_GET and DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET to get information about the image about to be transferred.
- The application initiates the transfer using a DG_CONTROL / DAT_IMAGExxxxFER / MSG_GET operation. The transfer occurs.
- Following a successful transfer, the Source returns TWRC_XFERDONE.
- The application sends the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation to acknowledge the end of the transfer and learn the number of pending transfers.

If the intent behind transferring a single image is to simply flush it from the Source (for example, an application may want to scan only every other page from a stack placed in a scanner with a document feeder), the following operation suffices:

- Issue a CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation. As with normal image transfer, this operation tells the Source that the application has completed acquisition of the current image, and the Source responds by reporting the number of pending transfers.

Preparing for Multiple Image Transfer

The DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation issued by the application at the end of every image transfer performs two important functions:

- It returns a count of pending transfers (in TW_PENDINGXFERS.Count)
- It transitions the session to State 6 (Transfer Ready) if the count of pending transfers is nonzero, or to State 5 (Source Enabled) if the count is zero. Recall that the count returned is a positive value if the Source knows the number of images available for acquisition. If the Source does not know the number of images available, the count returned us -1. The latter situation can occur if, for example, a document feeder is in use. Note that not knowing the number of images available includes the possibility that no further images are available; see the description of DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER for more on this.

We have just seen that after the MSG_ENDXFER operation is issued following an image transfer, the session is either in State 6 or State 5; that is, the session is still very much in an active state. If the session is in State 6 (i.e. “an image is available”), the application takes one of two actions so as to eventually transition the session to State 5 (i.e. “Source is ready to acquire an image, though none is available”):

- It continues to perform the single-image transfer process outlined earlier until no more images are available, or
- It issues a DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET to flush all pending transfers from the Source.

Once the session is back in State 5, the application has to decide whether to stay in State 5 or transition down to State 4 (“Source is open, and ready for capability negotiation”.) Two scenarios are possible here.

In one scenario, the application lets the Source control further state transitions. If the Source sends it a MSG_XFERREADY, the application restarts the multiple image transfer loop described above. If the Source sends it a MSG_CLOSEDSREQ (e.g. because the user activated the “Done” trigger on the UI displayed by the Source), the application sends back a DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS, thereby putting the session in State 4.

In the other scenario, the application directly controls session state transitions. For example, the application may want to shut down the current session as soon as the current batch of images have been transferred. In this case, the application issues a DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS as soon as the pending transfers count reaches zero.

It should be noted that there is no “right”, “wrong” or “preferred” scenario for an application to follow when deciding what to do once all images in the current set have been transferred. If an application wants to let the user control the termination of a session explicitly, it may well wait for the Source to send it a MSG_CLOSEDSREQ. On the other hand, the application may have a strong sense of what constitutes a session; for example, it may want to terminate a scan session as soon as a blank page is transferred. In such a case, the application will want to control the condition under which the MSG_DISABLEDS is sent.

Use of a Document Feeder

The term document feeder can refer to a physical device's automatic document feeder, such as might be available with a scanner, or to the logical feeding ability of an image database. Both input mechanisms apply although the following text uses the physical feeder for its discussion. The topics covered in this section are:

- Controlling whether to scan pages from the document feeder or the platen
- Detecting whether or not paper is ready for scanning
- Controlling scan lookahead

Note that these concepts are applicable to scanners that do not have feeders; see the discussion below for details.

Selecting the Document Feeder

Sometimes the use of a document feeder actually alters how the image is acquired. For instance, a scanner may move its light bar over a piece of paper if the paper is placed on a platen. When a document feeder is used, however, the same scanner might hold the light bar stable and scan the moving paper. To prepare for such variations the application and Source can explicitly agree to use the document feeder. The negotiation for this action must occur during State 4 **before** the Source is enabled using the following capability.

CAP_FEEDERENABLED

Determine if a Source has a document feeder available and, if so, select that option.

- To determine if this capability is supported, use a DG_CONTROL / DAT_CAPABILITY / MSG_GET operation. TWRC_FAILURE / TWCC_CAPUNSUPPORTED indicates this Source does not have the ability to select the document feeder.
- If supported, use the DG_CONTROL / DAT_CAPABILITY / MSG_SET operation during State 4.
- Set TW_CAPABILITY.Cap to CAP_FEEDERENABLED.
- Create a container of type TW_ONEVALUE and set it to TRUE. Reference TW_CAPABILITY.hContainer to the container.
- Execute the MSG_SET operation and check the Return Code.

  If TWRC_SUCCESS then the feeder is available and your request to use it was accepted. The application can now set other document feeder capabilities.

  If TWRC_FAILURE and TWCC_CAPUNSUPPORTED, TWCC_CAPBADOPERATION, or TWCC_BADVALUE then this Source does not have a document feeder capability or does not allow it to be selected explicitly.

Note: If an application wanted to prevent the user from using a feeder, the application should use a MSG_SET operation to set the CAP_FEEDERENABLED capability to FALSE.

Detecting Whether an Image is Ready for Acquisition

Having an image ready for acquisition in the Source device is independent of having a selectable document feeder. There are three possibilities here:

- The Source cannot tell whether an image is available,
- An image is available for acquisition, or
- No image is available for acquisition

These cases can be detected by first determining whether a Source can tell that image data is available for acquisition (case 1. vs. cases 2. and 3.) and then determining whether image data is available (case 2. vs. case 3.)The capabilities used to do so are as follows:

CAP_PAPERDETECTABLE

First, determine if the Source can tell that documents are loaded.

- To check if a Source can detect documents, use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation.
- Set the TW_CAPABILITY.Cap field to CAP_PAPERDETECTABLE.
- The Source returns TWRC_SUCCESS with the hContainer structure's value set to TRUE if it can detect a loaded document that is ready for acquisition. If the result code is TWRC_FAILURE with TWCC_CAPUNSUPPORTED or TWCC_BADVALUE, then the Source cannot detect that paper is loaded.

Note: CAP_PAPERDETECTABLE can be used independently of CAP_FEEDERENABLED. Also, an automatic document feeder need not be present for a Source to support this capability; e.g. a scanner that can detect paper on its platen should return TRUE.

The application cannot set this capability. The Source is simply reporting on a condition.

CAP_FEEDERLOADED

Next, determine if there are documents loaded in the feeder.

- To check if pages are present, use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation.
- Set the TW_CAPABILITY.Cap field to CAP_FEEDERLOADED.
- The Source returns TRUE if there are documents loaded. The information is in the container structure pointed to by the hContainer field of the TW_CAPABILITY structure.

Note: Neither CAP_FEEDERENABLED nor CAP_PAPERDETECTABLE need be TRUE to use this capability. A FALSE indication from this capability simply indicates that the feeder is not loaded or that the Source's feeder cannot tell. For a definitive answer, be sure to check CAP_PAPERDETECTABLE.

Controlling Scan Lookahead

With low-end scanners there is usually ample time for the CPU handling the image acquisition to process incoming image data on-the-fly or in the scan delay between pages. However, with higher performance scanners the CPU image processing time for a given page can become a significant fraction of the scan time. This problem can be alleviated if the scanner can scan ahead image data that the CPU has yet to acquire. This data can be buffered in scanner-local memory, or stored in main memory by the Source via a DMA operation while the CPU processes the current image.

Scan look-ahead is not always desirable, however. This is because the decision to continue a scan may be determined by the results of previously scanned images. For example, a scanning application may decide to stop a scan whenever it sees a blank page. If scan look-ahead were always enabled, one or more pages past the blank page may be scanned and transferred to the scanner's output bin. Such behavior may be incorrect from the point of view of the application's design

We have argued that the ability to control scan look-ahead is highly desirable. However, a single "enable scan look-ahead" command is insufficient to capture the richness of function provided by some scanners. In particular, TWAIN's model of document feeding has each image (e.g., sheet of paper) transition through a three stage process.

1. **Image is in input bin.** This action is taken by the user (for example, by placing a stack of paper into an auto-feeder.)
2. **Image is ready for scan.** This action causes the next available image to be placed at the start of the scan area. Set the CAP_AUTOFEED capability(described below)to automatically feed images to the start of the scan area.
3. **Image is scanned.** This action actually causes the image to be scanned. For example, the DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET operation initiates image transfer to an application via buffered memory. TWAIN allows a Source to pre-fetch images into Source-local memory (even before the application requests them) by setting the CAP_AUTOSCAN capability.

CAP_AUTOFEED

Enable the Source's automatic document feeding process.

- Use DG_CONTROL / DAT_CAPABILITY / MSG_SET.
- Set the TW_CAPABILITY.Cap field to CAP_AUTOFEED and set the capability to TRUE.
- When set to TRUE, the behavior of the Source is to eject one page and feed the next page after all frames on the first page are acquired. This automatic feeding process will continue whenever there is image data ready for acquisition (and the Source is in an enabled state). CAP_FEEDERLOADED is TRUE showing that pages are in the document feeder.

Note: CAP_FEEDERENABLED must be set to TRUE to use this capability. If not, the Source should return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

CAP_AUTOSCAN

Enable the Source’s automatic document scanning process.

- Use DG_CONTROL / DAT_CAPABILITY / MSG_SET.
- Set the TW_CAPABILITY.Cap field to CAP_AUTOSCAN and set the capability to TRUE.
- When set to TRUE, the behavior of the Source is to eject one page and scan the next page after all frames on the first page are acquired. This automatic scanning process will continue whenever there is image data ready for acquisition (and the Source is in an enabled state.

Note: Setting CAP_AUTOSCAN to TRUE implicitly sets CAP_AUTOFEED to TRUE also.

When your application uses automatic document feeding:

- Set CAP_XFERCOUNT to -1 indicating your application can accept multiple images.
- Expect the Source to return the TW_PENDINGXFERS.Count as -1. It indicates the Source has more images to transfer but it is not sure how many.
- Using automatic document feeding does not change the process of transferring multiple documents described earlier and in Chapter 3.

Control of the Document Feeding by the Application

In addition to automatic document feeding, TWAIN provides an option for an application to manually control the feeding of documents. This is only possible if the Source agrees to negotiate the following capabilities during States 5 and 6 by use of CAP_EXTENDEDCAPS. If CAP_AUTOFEED is set to TRUE, it can impact the way the Source responds to the following capabilities as indicated below.

CAP_FEEDPAGE

- If the application sets this capability to TRUE, the Source will eject the current page (if any) and feed the next page.
- To work as described requires that CAP_FEEDERENABLED and CAP_FEEDERLOADED be TRUE.
- If CAP_AUTOFEED is TRUE, the action is the still the same.
- The page ejected corresponds to the image that the application is acquiring (or is about to acquire). Therefore, if CAP_AUTOSCAN is TRUE and one or more pages have been scanned speculatively, the page ejected may correspond to a page that has already been scanned into Source-local buffers.

CAP_CLEARPAGE

- If the application sets this capability to TRUE, the Source will eject the current page and leave the feeder acquire area empty (that is, with no image ready to acquire).
- To work as described, this requires that CAP_FEEDERENABLED be TRUE and there be a paper in the feeder acquire area to begin with.
- If CAP_AUTOFEED is TRUE, the next page will advance to the acquire area.
- If CAP_AUTOSCAN is TRUE, setting this capability returns TWRC_FAILURE with TWCC_BADVALUE.

CAP_REWINDPAGE

- If the application sets this capability to TRUE, the Source will return the current page to the input area and return the last page from the output area into the acquisition area.
- To work as described requires that CAP_FEEDERENABLED be TRUE.
- If CAP_AUTOFEED is TRUE, the normal automatic feeding will continue after all frames of this page are acquired.
- The page rewound corresponds to the image that the application is acquiring. Therefore, if CAP_AUTOSCAN is TRUE and one or more pages have been scanned speculatively, the page rewound may correspond to a page that has already been scanned into Source-local buffers.

Transfer of Compressed Data

When using the Buffered Memory mode for transferring images, some Sources may support the transfer of data in a compressed format.

To determine if a Source supports transfer of compressed data and to set the capability

1. Use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation.
2. Set the TW_CAPABILITY.Cap field to ICAP_COMPRESSION.
3. The Source returns information about the compression schemes they support in the container structure pointed to by the hContainer field of TW_CAPABILITY. The identifiers for the compression alternatives all begin with TWCP_, such as TWCP_PACKBITS, and can be seen in the Constants section of Chapter 8 and in the TWAIN.H file.
4. If you wish to negotiate for the transfer to use one of the compression schemes shown, use the DG_CONTROL / DAT_CAPABILITY / MSG_SET operation.

The TW_IMAGEMEMXFER structure is used with the DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET operation. The structure looks like this:

```
typedef struct {
  TW_UINT16   Compression; /* A TWCP_xxxx constant */
  TW_UINT32   BytesPerRow;
  TW_UINT32   Columns;
  TW_UINT32   Rows;
  TW_UINT32   XOffset;
  TW_UINT32   YOffset;
  TW_UINT32   BytesWritten;
  TW_MEMORY   Memory;
} TW_IMAGEMEMXFER, FAR *pTW_IMAGEMEMXFER;
```

When compressed strips of data are transferred:

- The BytesPerRow field will be set to 0. The Columns, Rows, XOffset, and YOffset fields will contain TWON_DONTCARE32 indicating the fields hold invalid values. (The original image height and width are available by using the DG_IMAGE / DAT_IMAGEINFO / MSG_GET operation during State 6 prior to the transfer.)
- Transfer buffers are always completely filled by the Source. For compressed data, it is very likely that at least one partial line will be written into the buffer.
- The application is responsible for deallocating the buffers.

When compressed, tiled data are transferred:

- All fields in the structure contain valid data. BytesPerRow, Columns, Rows, XOffset, and YOffset all describe the uncompressed tile. Compression and BytesWritten describe the compressed tile.
- In this case, unlike with compressed, strip data transfer, the Source allocates the transfer buffers. This allows the Source to create buffers of differing sizes so that complete, compressed tiles can be transferred to the application intact (not split between sequential buffers). Under these conditions, the application should set the fields of the TW_MEMORY structure so Flags is TWMF_DSOWNS, Length is TWON_DONTCARE32 and TheMem is NULL. The Source must assume that the application will keep the previous buffer rather than releasing it. Therefore, the Source must allocate a new buffer for each transfer.
- The application is responsible for deallocating the buffers.
- Finally, the application cannot assume that the tiles will be transferred in any particular, logical order.

JPEG Compression

TWAIN supports transfer of several forms of compressed data. JPEG compression is one of them. The JPEG compression algorithm provides compression ratios in the range of 10:1 to 25:1 for grayscale and full-color images, often without causing visible loss of image quality. This compression, which is created by the application of a series of “perceptual” filters, is achieved in three stages:

Color Space Transformation and Component Subsampling (Color Images Only, Not for Grayscale)

The human eye is far more sensitive to light intensity (luminance) than it is to light frequency (chrominance, or “color”) since it has, on average, 100 million detectors for brightness (the “rods”) but only about 6 million detectors for color (the “cones”). Substantial image compression can be achieved simply by converting a color image into a more efficient luminance/chrominance color space and then subsampling the chrominance components.

This conversion is provided for by the TW_JPEGCOMPRESSION structure. By specifying the TW_JPEGCOMPRESSION.ColorSpace = TWPT_YUV, Source RGB data is converted into more space-efficient YUV data (better known as CCIR 601-1 or YCbCr). TW_JPEGCOMPRESSION.SubSampling specifies the ratio of luminance to chrominance samples in the resulting YUV data stream, and a typical choice calls for two luminance samples for every chrominance sample. This type of subsampling is specified by entering 0x21102110 into the TW_JPEGCOMPRESSION.SubSampling field. A larger ratio of four luminance samples for every chrominance sample is represented by 0x41104110. To sample two luminance values for every chrominance sample in both the horizontal and vertical axes, use a value of 0x21102110.

Application of the Discrete Cosine Transform (DCT) and Quantization

The original components (with or without color space conversion) are next mathematically converted into a spatial frequency representation using the DCT and then filtered with quantization matrices (each frequency component is divided by its corresponding member in a quantization matrix). The quantization matrices are specified by TW_JPEGCOMPRESSION.QuantTable[] and up to four quantization matrices may be defined for up to four different original components. TW_JPEGCOMPRESSION.QuantMap[] maps the particular original component to its respective quantization matrix.

Note: Defaults are provided for the quantization map and tables are suggested in Section K of the JPEG Draft International Standard, version 10918-1 are used as the default tables for QuantTable, HuffmanDC, and HuffmanAC by TWAIN. The default tables are selected by putting NULL into each of the TW_JPEGCOMPRESSION.QuantTable[] entries.

Huffman encoding

The resulting coefficients from the DCT and quantization steps are further compressed in one final stage using a loss-less compression algorithm called Huffman encoding. Application developers can provide Huffman tables, though typically the default tables—selected by writing NULL into TW_JPEGCOMPRESSION.HuffmanDC[] and TW_JPEGCOMPRESSION.HuffmanAC[]—yield very good results.

The algorithm optionally supports the use of restart marker codes. The purpose of these markers is to allow random access to strips of compressed data in JPEG data stream. They are more fully described in the JPEG specification.

See Chapter 8 for the definition of the TW_JPEGCOMPRESSION data structure. Example data structures are shown below for RGB image compression and grayscale image compression:

```
/* RGB image compression - YUV conversion and 2:1:1 chrominance */
   /* subsampling                                                */

   typedef struct TW_JPEGCOMPRESSION  myJPEG;

   myJPEG.ColorSpace       = TWPT_YUV;        // convert RGB to YUV
   myJPEG.SubSampling      = 0x21102110;      // 2 Y for each U, V
   myJPEG.NumComponents    = 3;               // Y, U, V
   myJPEG.RestartFrequency = 0;               // No restart markers
   myJPEG.QuantMap[0]      = 0;               // Y component uses table0
   myJPEG.QuantMap[1]      = 1;               // U component uses table 1
   myJPEG.QuantMap[2]      = 1;               // V component uses table 1
   myJPEG.QuantTable[0]    = NULL;            // select defaults for quant
                                              // tables
   myJPEG.QuantTable[1]    = NULL;            //
   myJPEG.QuantTable[2]    = NULL;            //
   myJPEG.HuffmanMap[0]    = 0;               // Y component uses DC & AC
                                              // table 0
   myJPEG.HuffmanMap[1]    = 1;               // U component uses DC & AC
                                              // table 1
   myJPEG.HuffmanMap[2]    = 1;               // V component uses DC & AC
                                              // table 1
   myJPEG.HuffmanDC[0]     = NULL;            // select default for Huffman
                                              // tables
   myJPEG.HuffmanDC[1]     = NULL;            //
   myJPEG.HuffmanAC[0]     = NULL;            //
   myJPEG.HuffmanAC[1]     = NULL;            //

   /* Grayscale image compression - no color space conversion or  */
   /* subsampling                                                 */

   typedef struct TW_JPEGCOMPRESSION  myJPEG;

   myJPEG.ColorSpace       = TWPT_GRAY;       // Grayscale data
   myJPEG.SubSampling      = 0x10001000;      // no chrominance components
   myJPEG.NumComponents    = 1;               // Grayscale
   myJPEG.RestartFrequency = 0;               // No restart markers
   myJPEG.QuantMap[0]      = 0;               // select default for quant
                                              // map
   myJPEG.QuantTable[0]    = NULL;            //
   myJPEG.HuffmanMap[0]    = 0;               // select default for Huffman
                                              // tables
   myJPEG.HuffmanDC[0]     = NULL;            //
   myJPEG.HuffmanAC[0]     = NULL;            //
```

The resulting compressed images from these examples will be compatible with the JPEG File Interchange Format (JFIF version 1.1) and will therefore be usable by a variety of applications that are JFIF-aware.

Alternative User Interfaces

Alternatives to Using the Source Manager's Select Source Dialog

TWAIN ships its Source Manager code to act as the communication vehicle between application and Source. One of the services the Source Manager provides is locating all available Sources that meet the application's requirements and presenting those to the user for selection.

It is recommended that the application use this approach. However, the application is not required to use this service. Two alternatives exist:

- The application can develop and present its own custom selection interface to the user. This is presented in response to the user choosing **Select Source...** from its menu.
- Or, if the application is dedicated to control of a specific Source, the application can transparently select the Source. In this case, the application does not functionally need to have a Select Source... option in the menu but a grayed-out one should be displayed for consistency with all other TWAIN-compliant applications.

Displaying a custom selection interface:

1. Use the DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST operation to have the Source Manager locate the first Source available. The name of the Source is contained in the TW_IDENTITY.ProductName field. Save the TW_IDENTITY structure.
2. Use the DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT to have the Source Manager locate the next Source. Repeatedly use this operation until it returns TWRC_ENDOFLIST indicating no more Sources are available. Save the TW_IDENTITY structure.
3. Use the ProductName information to display the choices to the user. Once they have made their selection, use the saved TW_IDENTITY structure and the DG_CONTROL / DAT_IDENTITY / MSG_OPENDS operation to have the Source Manager open the desired Source. (Note, using this approach, as opposed to the MSG_USERSELECT operation, the Source Manager does not update the system default Source information to reflect your choice.)

Transparently selecting a Source:

If the application wants to open the system default Source , use the DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT operation to have the Source Manager locate the default Source and fill the TW_IDENTITY structure with information about it. The name of the Source is contained in the TW_IDENTITY.ProductName field. Save the TW_IDENTITY structure.

OR

If you know the ProductName of the Source you wish to use and it is not the system default Source, use the DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST and DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT operations to have the Source Manager locate each Source. You must continue looking at Sources until you verify that the desired Source is available. Save the TW_IDENTITY structure when you locate the Source you want. If the Return Code TWRC_ENDOFLIST appears before the desired Source is located, it is not available.

Use the saved TW_IDENTITY structure and the DG_CONTROL / DAT_IDENTITY / MSG_OPENDS operation to have the Source Manager open the desired Source. (Note, using this approach, rather than MSG_USERSELECT, the Source Manager does not update the system default Source information to reflect your choice.)

Alternatives to Using the Source's User Interface

Just as with the Source Manager's Select Source dialog, the application may ask to not use the Source's user interface. Certain types of applications may not want to have the Source's user interface displayed. An example of this can be seen in some text recognition packages that wish to negotiate a few capabilities (i.e. pixel type, resolution, page size) and then proceed directly to acquiring and transferring the data.

To Enable the Source without Displaying its User Interface

- Use the DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS operation.
- Set the ShowUI field of the TW_USERINTERFACE structure to FALSE.
- When the command is received and accepted (TWRC_SUCCESS), the Source does not display a user interface but is armed to begin capturing data. For example, in a flatbed scanner, the light bar will light and begin to move. A handheld scanner will be armed and ready to acquire data when the "go" button is pressed on the scanner. Other devices may respond differently but they all will either begin acquisition immediately or be armed to begin acquiring data as soon as the user interacts with the device.

Capability Negotiation is Essential when the Source's User Interface is not Displayed

- Since the Source's user interface is not displayed, the Source will not be giving the user the opportunity to select the information to be acquired, etc. Unless default values are acceptable, current values for all image acquisition and control parameters must be negotiated before the Source is enabled, i.e. while the session is in State 4.

When TW_USERINTERFACE.ShowUI is set to FALSE:

- The application is still required to pass all events to the Source (via the DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation) while the Source is enabled.
- The Source must display the minimum possible user interface containing only those controls required to make the device useful in context. In general, this means that no user interface is displayed, however certain devices may still require a trigger to initiate the scan.
- The Source still displays a progress indicator during the acquisition. The application can suppress this by setting CAP_INDICATORS to FALSE, if the Source allows this.
- The Source still displays errors and other messages related to the operation of its device. This cannot be turned off.
- The Source still sends the application a MSG_XFERREADY notice when the data is ready to be transferred.
- The Source may or may not send a MSG_CLOSEDSREQ to the application asking to be closed since this is often user-initiated. Therefore, after the Source has returned to State 5 (following the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation and the TW_PENDINGXFERS.Count = 0), the application can send the DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS operation.

Note: Some Sources may display the UI even when ShowUI is set to FALSE. An application can determine whether ShowUI can be set by interrogating the CAP_UICONTROLLABLE capability. If CAP_UICONTROLLABLE returns FALSE but the ShowUI input value is set to FALSE in an activation of DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS, the enable DS operation returns TWRC_CHECKSTATUS but displays the UI regardless. Therefore, an application that requires that the UI be disabled should interrogate CAP_UICONTROLLABLE before issuing MSG_ENABLEDS.

Modal Versus Modeless User Interfaces

The Source Manager's user interface is a modal interface but the Source may provide a modeless or modal interface. Here are the differences:

Modeless

When a Source uses a modeless user interface, although the Source's interface is displayed, the user is still able to access the application by clicking on the application's window and making it active.

The user is expected to click on a Close button on the Source's user interface when they are ready for that display to go away. The application must NOT automatically close a modeless Source after the first (or any subsequent) transfer, even if the application is only interested in receiving a single transfer. If the application closes the Source before the user requests it, the user is likely to become confused about why the window disappeared. Wait until the user indicates the desire to close the Source's window and the Source sends this request (MSG_CLOSEDSREQ) to the application before closing the Source.

Modal

A Source using a modal user interface prevents the user from accessing other windows.

For Windows only, if the interface is application modal, the user cannot access other applications but can still access system utilities. If the interface is system modal (which is rare), the user cannot access anything else at an application or system level. A system modal dialog might be used to display a serious error message, like a UAE.

If using a modal interface, the Source can perform only one acquire during a session although there may be multiple frames per acquisition. The Source will send a close request to the application following the completion of the data transfer. Again, the application waits to receive this request.

The Source indicates if it is using a modeless or modal interface after the application enables it using the DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS operation. The data structure used in the operation (TW_USERINTERFACE) contains a field, ShowUI, which is set by the application to indicate whether the Source should display its user interface. If the application requests the user interface be shown, the Source sets the ModalUI field to indicate if its user interface is modal (TRUE) or modeless (FALSE).

When requested by the Source, the application uses the DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS operation to remove the Source's user interface.

Grayscale and Color Information for an Image

There are operation triplets in TWAIN that allow the application developer to interact with and influence the grayscale or color aspect of the images that a Source transfers to the application. The following operations provide these abilities:

CIE Color Descriptors

DG_IMAGE / DAT_CIECOLOR / MSG_GET

Grayscale Changes

DG_IMAGE / DAT_GRAYRESPONSE / MSG_RESET
DG_IMAGE / DAT_GRAYRESPONSE / MSG_SET

Palette Color Data

DG_IMAGE / DAT_PALETTE8 / MSG_GET
DG_IMAGE / DAT_PALETTE8 / MSG_GETDEFAULT
DG_IMAGE / DAT_PALETTE8 / MSG_RESET
DG_IMAGE / DAT_PALETTE8 / MSG_SET

RGB Response Curve Data

DG_IMAGE / DAT_RGBRESPONSE / MSG_RESET
DG_IMAGE / DAT_RGBRESPONSE / MSG_RESET

CIE Color Descriptors

The CIE XYZ approach is a method for storing color data which simplifies doing mathematical manipulations on the data. (The topic of CIE XYZ color space is discussed thoroughly in Appendix A.)

If your application wishes to receive the image data in this format:

1. You must ensure that the Source is able to provide data in CIE XYZ format. To check this, use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation and get information on the ICAP_PIXELTYPE. If TWPT_CIEXYZ is returned as one of the supported types, the Source can provide data in CIE XYZ format.
2. After verifying that the Source supports it, the application can specify that data transfers should use the CIE XYZ format by invoking a DG_CONTROL / DAT_CAPABILITY / MSG_SET operation on the ICAP_PIXELTYPE. Use a TW_ONEVALUE container whose value is TWPT_CIEXYZ.

To determine the parameters that were used by the Source in converting the color data into the CIE XYZ format, use the DG_IMAGE / DAT_CIECOLOR / MSG_GET operation following the transfer of the image.

Grayscale Changes

(The grayscale operations assume that the application has instructed the Source to provide grayscale data by setting its ICAP_PIXELTYPE to TWPT_GRAY and the Source is capable of this.)

The application can request that the Source apply a transfer curve to its grayscale data prior to transferring the data to the application. To do this, the application uses the DG_IMAGE / DAT_GRAYRESPONSE / MSG_SET operation. The desired transfer curve information is placed by the application within the TW_GRAYRESPONSE structure (the actual array is of type TW_ELEMENT8). The application must be certain to check the Return Code following this request. If the Return Code is TWRC_FAILURE and the Condition Code shows TWCC_BADPROTOCOL, this indicates the Source does not support grayscale response curves (despite supporting grayscale data).

If the Source allows the application to set the grayscale transfer curve, there must be a way to reset the curve to its original non-altered value. Therefore, the Source must have an “identity response curve” which does not alter grayscale data but transfers it exactly as acquired. When the application sends the DG_IMAGE / DAT_GRAYRESPONSE / MSG_RESET operation, the Source resets the grayscale response curve to its identity response curve.

Palette Color Data

(The palette8 operations assume that the application has instructed the Source to use the TWPT_PALETTE type for its ICAP_PIXELTYPE and that the Source has accepted this.)

The DAT_PALETTE8 operations allow the application to inquire about a Source’s support for palette color data and to set up a palette color transfer. The operations are specialized for 8-bit data, whether grayscale or color (8-bit or 24-bit). The MSG_GET operation allows the application to learn what palette was used by the Source during the image acquisition. The application should always execute this operation immediately after an image transfer rather than before because the Source may optimize the palette during the acquisition process. Some Sources may allow an application to define the palette to be used during image acquisition via the MSG_SET operation. Be sure to check the Return Code to verify that it is TWRC_SUCCESS following a MSG_SET operation. That is the only way to be certain that your requested palette will actually be used during subsequent palette transfers.

RGB Response Curve Data

(The RGB Response curve operations assume that the application has instructed the Source to provide RGB data by setting its ICAP_PIXELTYPE to TWPT_RGB and the Source is capable of this.)

The application can request that the Source apply a transfer curve to its RGB data prior to transferring the data to the application. To do this, the application uses the DG_IMAGE / DAT_RGBRESPONSE / MSG_SET operation. The desired transfer curve information is placed by the application within the TW_RGBRESPONSE structure (the actual array is of type TW_ELEMENT8). The application must be certain to check the Return Code following this request. If the Return Code is TWRC_FAILURE and the Condition Code shows TWCC_BADPROTOCOL, this indicates the Source does not support RGB response curves (despite supporting RGB data).

If the Source allows the application to set the RGB response curve, there must be a way to reset the curve to its original non-altered value. Therefore, the Source must have an “identity response curve” which does not alter RGB data but transfers it exactly as acquired. When the application sends the DG_IMAGE / DAT_RGBRESPONSE / MSG_RESET operation, the Source resets the RGB response curve to its identity response curve.

Contrast, Brightness, and Shadow Values

There is considerable confusion about what is the appropriate way to present these actual features for a particular device. Anyone who has attempted to support these capabilities knows that the recommended ranges do not accurately reflect the capabilities of real world devices. Data source developers have tried many different methods of getting the correct response for their data source, and not all are consistent.

By providing a meaningful step size, or by providing a different container, a data source can provide the application with enough information to accurately model the actual ability of the device. For an application that wishes to present a custom User Interface for this type of capability, it is not really useful to the user if it provides 2000 steps from -1000 to +1000, especially if the device really only supports a small number of levels.

Since both data source developers and application developers read the same specification, it can be assumed that it is not acceptable to provide values that do not fit within the documented ranges for these types of capabilities.

The following suggestion is an example of how to follow the specification, and provide the most accurate values for the particular data source.

Example 1: ICAP_BRIGHTNESS Supporting Only Three Levels

The specification requirement stated in Chapter 9 is as follows:

> *"Source should normalize the values into the range. Make sure that a '0' value is available as the Current Value when the Source starts up. If the Source's ± range is asymmetric about the '0' value, set range maxima to ±1000 and scale homogeneously from the '0' value in each direction. This will yield a positive range whose step size differs from the negative range's step size."*

Note: It should be expanded in this statement that for a step size that differs in the negative and positive range, a TW_ENUMERATION container must be used. A TW_RANGE container is not suitable for representing a non-linear step size).

Assume the actual device simply supports the options normal, lighten, and darken. These can fit into the constraints by mapping actual values to required values:

Normal = 0
Lighten = -1000
Darken = 1000

These values can be placed in a TW_RANGE container with a step size of 1000, or into a TW_ENUMERATION containing only the 3 values. { -1000, 0, 1000 }, the current and default values are 0.

5 Source Implementation

Companies that produce image-acquisition devices, and wish to gain the benefits of being TWAIN-compliant, must develop TWAIN-compliant Source software to drive their device. The Source software is the interface between TWAIN's Source Manager and the company's physical (or logical) device. To write effective Source software, the developer must be familiar with the application's expectations as discussed in the other chapters of this document. This chapter discusses:

Chapter Contents

The Structure of a Source

The following sections describe the structure of a source.

On Windows

Implementation

The Source is implemented as a Dynamic Link Library (DLL). Sources should link to TWAIN.LIB at link time. The Source will not run stand-alone. The DLL typically runs within the (first) calling application's heap although DLLs may be able to allocate their own heap and stack space. There is only one copy of the DLL's code and data loaded at run-time, even if more than one application accesses the Source. For more information regarding DLLs on Win32s, Windows95, and Windows NT please refer to Microsoft documents.

Naming and Location

The DLL's file name must end with a .DS extension. The Source Manager recursively searches for your Source in the TWAIN sub-directory of the Windows directory (typically C:\WINDOWS on Windows 95/98, or C:\WINNT on Windows NT). To reduce the chance for naming collisions, each Source should create a sub-directory beneath TWAIN, giving it a name relevant to their product. The Source DLLs are placed there. Supporting files may be placed there as well, but since this is a system directory which may only be modifiable by the System Administrator, Sources must not write any information into this directory after the installation.

Entry Points and Segment Attributes

Every Source is required to have a single entry point called DS_Entry (see Chapter 6). For 16-bit sources only, in order to speed up the Source Manager's ability to identify Sources, the Source entry point DS_Entry( ) and the code to respond to the DG_CONTROL / DAT_IDENTITY / MSG_GET operation must reside in a segment marked as PRELOAD. All other segments should be marked as LOADONCALL (with the exception of any interrupt handler that may exist in the Source which needs to be in a FIXED code segment).

Resources

- **Version Information -** The Microsoft VER.DLL is included with the TWAIN toolkit for use by your installation program, if you have one, to validate the version of the currently installed Source Manager. Sources should be marked with the Version information capability defined in Microsoft Windows 3.1. To do this, you can use the resource compiler from the version 3.1 SDK. VER.DLL and the version stamping are also compatible with Microsoft Windows version 3.0.
- **Icon Id -** Future versions of the TWAIN Source Manager may display the list of available Sources using a combination of the ProductName (in the Source's TW_IDENTITY structure) and an Icon (as the Macintosh version currently does). Therefore, it is recommended that you add this icon into your Source resource file today. This will allow your Source to be immediately compatible with any upcoming changes. The icon should be identified using TWON_ICONID from the TWAIN.H file.

General Notes

- **GlobalNotify -** Microsoft Windows allows only one GlobalNotify handler per task. As the Source resides in the application heap, the Source should **not** use the GMEM_NOTIFY flag on the memory blocks allocated as this may disrupt the correct behavior of the application that uses GlobalNotify.
- **Windows Exit Procedure (WEP) -** During the WEP, the Source is being unloaded by Microsoft Windows. The Source should make sure all the resources it allocated and owns get released whether or not the Source was terminated properly.

On Macintosh

Implementation

A Source on a Macintosh is implemented as a Code Resource. The Source will not run stand-alone. The Source's code will exist and run within the calling application's heap. A separate copy of the Source's code will be made for each application that opens the Source. Macintosh development books such as *Inside Macintosh* describe the special requirements for developing Code Resources.

A Source can communicate between various running copies of itself through its resource file. If a Source must enforce a maximum number of applications that the device it controls can support, for instance, it can maintain the number of current connects within the Source's resource fork. If the maximum number of connects is exceeded when a new application tries to open a new copy of the Source, the Source should display an appropriate error message to the user and then return TWRC_FAILURE with a Condition Code of TWCC_MAXCONNECTIONS. This gives each running copy visibility of the total number of connections. Other information that needs to be communicated between instances of the Source can use this same method.

Naming and Location

The type for a Source is DSRC. The Source Manager will recursively search for files of this type in the TWAIN folder. The creator is vendor-dependent.

Under System 6.x, Sources are located within the TWAIN folder which resides within the System Folder. It is recommended that each Source file, along with any other files it may require, be installed into a uniquely named folder within the TWAIN folder. Placing your files within a specially named folder will limit the chances of name collisions of the Source's support files (or the Source itself) with the names of other Sources and Source-support files already installed. The Source Manager will recursively search out all Sources within the TWAIN folder.

Under System 7, the TWAIN folder is located within the Preferences folder (which resides within the System Folder). All Sources should be located within the Preferences folder per the rules of the preceding paragraph. The Source Manager can sense the version of the operating system and will look in the correct folder.

Entry Point

The entry point for the code resource, DS_Entry, **must** be the first address in the resource and **must** have the format described in Chapter 6.

Resources

- **Version Information -** The Macintosh Source Manager will keep its version information in it **vers** resource. The application can read this resource for Source Manager information. The resource ID of this resource is in the constant TWON_DSMID (see the TWAIN.H file).
- **Code Resources -** The initial code resource that will be loaded by the Source Manager must have a resource type of DSRC. This resource must have a resource ID defined in the constant TWON_DSRCID (see the TWAIN.H file). The first address in this resource must be DS_Entry, or must call it directly.
- **ICN# Resource -** The icon that is displayed by the Source Manager in the Select Source... dialog is an ICN# resource with a resource ID defined in the constant TWON_ICONID (see the TWAIN.H file).

Operation Triplets

In Chapter 3, we introduced all of the triplets that an application can send to the Source Manager or ultimately to a Source. There are several other triplet operations which do not originate from the application. Instead, they originate from the Source Manager or Source and are introduced in this chapter. All defined operation triplets are listed in detail in Chapter 7.

Triplets from the Source Manager to the Source

There are three operation triplets that are originated by the Source Manager. They are:

DG_CONTROL / DAT_IDENTITY

MSG_GET	Returns the Source’s identity structure
MSG_OPENDS	Opens and initializes the Source
MSG_CLOSEDS	Closes and unloads the Source

The DG_CONTROL / DAT_IDENTITY / MSG_GET operation is used by the Source Manager to identify available Sources. It may send this operation to the Source **at any time** and the Source **must be prepared** to respond informatively to it. That means, the Source must be able to return its identity structure before being opened by the Source Manager (with the MSG_OPENDS command). The Source’s initially loaded code segment must be able to perform this function without loading any additional code segments. This allows quick identification of all available Sources and is the only operation a Source must support before it is formally opened.

The TW_IDENTITY structure looks like this:

```
typedef struct {
      TW_UINT32     Id;
      TW_VERSION    Version;
      TW_UINT16     ProtocolMajor;
      TW_UINT16     ProtocolMinor;
      TW_UINT32     SupportedGroups;
      TW_STR32      Manufacturer;
      TW_STR32      ProductFamily;
      TW_STR32      ProductName;
} TW_IDENTITY, FAR *pTW_IDENTITY;
```

The ProductName field in the Source's TW_IDENTITY structure should uniquely identify the Source. This string will be placed in the Source Manager's Select Source... dialog for the user. (The file name of the Source should also approximate the ProductName, if possible, to add clarity for the user at installation time.) Fill in all fields except the Id field which will be assigned by the Source Manager. The unique Id number that identifies your Source during its current session will be passed to your Source when it is opened by the MSG_OPENDS operation. Sources on Windows must save this TW_IDENTITY.Id information for use when sending notifications from the Source to the application.

Sources and the Event Loop

Handling Events

On both Windows and Macintosh, when a Source is enabled (i.e. States 5, 6, and 7), the application must pass all events (messages) to the Source. Since the Source runs subservient to the application, this ensures that the Source will receive all events for its window. The event will be passed in the TW_EVENT data structure that is referenced by a DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT command.

Note: Starting with TWAIN 1.8, it is now possible for events to be managed in State 4 only to support CAP_DEVICEEVENTS. This is a fundamental change from previous TWAIN behavior that has been added to allow the Source to notify the Application of important changes in the state of the Source even while in State 4. Note also that the default value for CAP_DEVICEEVENTS (if supported) must be an empty TW_ARRAY, indicating the event reporting is turned off. This is essential to allow backward compatibility with pre-1.8 Applications.

Routing all messages to all connected Sources while they are enabled places a burden on the application and creates a potential performance bottleneck. Therefore, the Source must process the incoming events as quickly as possible. The Source should examine each incoming operation before doing anything else. Only one operation's message field says MSG_PROCESSEVENT so always look at the message field first. If it indicates MSG_PROCESSEVENT then:

Immediately determine if the event belongs to the Source.

```
If it does
    Set the Return Code for the operation to TWRC_DSEVENT
    Set the TWMessage field to MSG_NULL
    Process the event
Else
    Set the Return Code to TWRC_NOTDSEVENT
    Set the TWMessage field to MSG_NULL
    Return to the application immediately
```

If the Source developer fails to process events with this high priority, the user may see degraded performance whenever the Source is frontmost which reflects poorly on the Source.

On Windows, the code fragment looks like the following:

```
TW_UINT16 CALLBACK DS_Entry(pTW_IDENTITY pSrc,
                                        TW_UINT32 DG,
                                        TW_UINT16 DAT,
                                        TW_UINT16 MSG,
                                        TW_MEMREF pData)
{
    TWMSG     twMsg;
    TW_UINT16 twRc;
    // Valid states: 5 - 7.  As soon as the application has enabled the
    // Source it must being sending the Source events.  This allows the
    // Source to receive events to update its user interface and to
    // return messages to the application. The app sends down ALL
    // message, the Source decides which ones apply to it.
if (MSG == MSG_PROCESSEVENT)
    {
          if (hImageDlg && IsDialogMessage(hImageDlg,
(LPMSG)(((pTW_EVENT)pData)->pEvent)))
        {
             twRc = TWRC_DSEVENT;

            // The source should, for proper form, return a MSG_NULL for
            // all Windows messages processed by the Data Source

            ((pTW_EVENT)pData)->TWMessage = MSG_NULL;
        }
```

```
            else
            {
                      // notify the application that the source did not
                      // consume this message
                      twRc = TWRC_NOTDSEVENT;
              ((pTW_EVENT)pData)->TWMessage = MSG_NULL;
            }
                 }
                 else
                 {
              // This is a Twain message, process accordingly.
              // The remainder of the Source's code follows...
                 }
    return twRc;
    }
```

The Windows IsDialogMessage( ) call is used in this example. Sources can also use other Windows calls such as TranslateAccelerator( ) and TranslateMDISYSAccel( ).

Communicating to the Application

As explained in Chapter 3, there are four instances where the Source must originate and transmit a notice to the application:

- **When it has data ready to transfer (MSG_XFERREADY)**

  The Source must send this message when the user clicks the "GO" button on the Source's user interface or when the application sends a DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS operation with ShowUI = FALSE. The Source will transition from State 5 to State 6. The application should now perform their inquiries regarding TW_IMAGEINFO and capabilities. Then, the application issues a DG_IMAGE / DAT_IMAGExxxxXFER / MSG_GET operation to begin the transfer process. Typically, though it is not required, it is at this time that a flatbed scanner (for example) will begin **simultaneously** to acquire and transfer the data using the specified transfer mode.

- **When it needs to have its user interface disabled (MSG_CLOSEDSREQ)**

  Typically, the Source will send this only when the user clicks on the "CLOSE" button on the Source's user interface or when an error occurs which is serious enough to require terminating the session with the application. The Source should be in (or transition to) State 5. The application should respond by sending a DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS operation to transition the session back to State 4.

- **When the user has pressed the OK button in a Source's dialog that was brought up with DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDSUIONLY (MSG_CLOSEDSOK).**

  Applications should use this event as the indicator that the user has set all the desired attributes from the Source's GUI.

When the Source needs to report a Device Event. Note that the application must explicitly request the Source to supply Device Events (MSG_DEVICEEVENT). Sources must only provide those Device Events requested by a Source through the CAP_DEVICEEVENT capability. The default for this capability when the Source starts up is an empty TW_ARRAY, indicating that no Device Events are being reported. Applications that turn on Device Events must issue a DG_CONTROL / DAT_DEVICEEVENT / MSG_GET command as soon as possible after receiving a Device Event.

These notices are sent differently on Windows versus Macintosh systems.

On Windows

The Source creates a call to DSM_Entry (the entry point in the Source Manager) and fills the destination with the TW_IDENTITY structure of the application. The Source uses one of the following triplets:

DG_CONTROL / DAT_NULL / MSG_XFERREADY
DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ

The Source Manager, on Windows, recognizes the notice and makes sure the message is received correctly by the application.

On Macintosh

The Source on Macintosh does not use the operations described above. Instead, it uses a TW_EVENT structure to send its notice to the application. The TW_EVENT structure is created by the application and sent to the Source as data in a DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation.

Normally, the Source places MSG_NULL in the TW_EVENT.TWMessage field. To relay the notice, the Source places one of the following in the TWMessage field:

MSG_XFERREADY
MSG_CLOSEDSREQ

The application examines that field when the DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation concludes and recognizes these two special notices from the Source.

User Interface Guidelines

Each TWAIN-compliant Source provides a user interface to assist the user in acquiring data from their device. Although each device has its own unique needs, the following guidelines are provided to increase consistency among TWAIN-compliant devices.

Displaying the User Interface

The application issues DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS to transition the session from State 4 to 5.

The TW_USERINTERFACE data structure contains these fields:

- **ShowUI -** If set to TRUE, the Source displays its user interface. If FALSE, the application will be providing its own.
- **hParent -** Used by Sources on Windows only. It indicates the application's window handle. This is to be designated as the Source's parent for the dialog so it is a proper child of its parent application.
- **ModalUI -** To be set by the Source to TRUE or FALSE.

Sources are not required to allow themselves to be enabled without showing their user interface (ShowUI = FALSE) but it is strongly recommended that they allow this. If your Source cannot be used without its user interface, it should enable showing the user interface (just as if ShowUI = TRUE) and return TWRC_CHECKSTATUS. All Sources, however, must report whether or not they honor ShowUI set to FALSE via the CAP_UICONTROLLABLE capability. This allows applications to know whether the Source-supplied user interface can be suppressed before it is displayed.

When the user interface is disabled (by DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS), a pointer to a TW_USERINTERFACE is included in the pData parameter.

Modal versus Modeless Interfaces

As stated in Chapter 4, the Source's user interface may be modal or modeless. The modeless approach gives the user more control and is recommended whenever practical. Refer to the information following this table for specifics about Windows and Macintosh implementation.

Error and Device Control Indicators

The Source knows what is happening with the device it controls. Therefore, the Source is responsible for determining when and what information regarding errors and device controls (ex. "place paper in document feeder") should be presented to the user. Error information should be placed by the Source on top of either the application's or Source's user interface. Do not present error messages regarding capability negotiation to the user since this should be transparent.

Progress Indicators

The Source should display appropriate progress indicators for the user regarding the acquisition and/or transfer processes. The Source must provide this information regardless of whether or not its user interface is displayed (ShowUI equals TRUE or FALSE). To suppress the indicators when the user interface is not displayed, the application should negotiate the CAP_INDICATORS capability to be FALSE.

Impact of Capability Negotiation

If the Source has agreed to limit the Available Values and/or to set the Current Value, the interface should reflect the negotiation. However, if a capability has not been negotiated by the application, the interface should not be modified (don't gray out a control because it wasn't negotiated.)

Advanced Topics

If a Source can acquire from more than one device, the Source should allow the user to choose which device they wish to acquire from. Provide the user with a selection dialog that is similar to the one presented by the Source Manager's Select Source... dialog.

Implementing Modal and Modeless User Interfaces

On Windows

You cannot use the modal dialog creation call DialogBox( ) to create the Source's user interface main window. To allow event processing by both the application and the Source, this call cannot be used. Modal user interfaces in Source are not inherently bad, however. If a modal user interface makes sense for your Source, use either the CreateDialog( ) or CreateWindow( ) call.

Modal (App Modal)

It is recommended that the Source's main user interface window be created with a modeless mechanism. Source writers can still decide to make their user interface behave modally if they choose. It is even appropriate for a very simple "click and go" interface to be implemented this way.

This is done by first specifying the application's window handle (hWndParent) as the parent window when creating the Source's dialog/window and second by enabling/disabling the parent window during the MSG_ENABLEDS / MSG_DISABLEDS operations. Use EnableWindow(hWndParent, FALSE) to disable the application window and EnableWindow(hWndParent, TRUE) to re-enable it.

Modeless

If implementing a modeless user interface, specify NULL as the parent window handle when creating the Source's dialog/window. Also, it is suggested that you call BringWindowToTop( ) whenever a second request is made by the same application or another application requesting access to a Source that supports multiple application connections.

On Macintosh

It is recommended that the Source's main user interface window be created with a modeless mechanism. Source writers can still decide to make their user interface behave modally if they choose. It is even appropriate for a very simple "click and go" interface to be implemented this way.

Capability Negotiation

Capability negotiation is a critical area for a Source because it allows the application to understand and influence the images that it receives from your Source.

Inquiries From the Application

While the Source is open but not yet enabled (from DG_CONTROL / DAT_IDENTITY / MSG_OPENDS until DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS), the application can inquire the values of all supported capabilities, and request to set those values.

Once the Source is enabled, the application may only **inquire** about capabilities. An attempt to set a capability should fail with TWRC_FAILURE and TWCC_SEQERROR (unless CAP_EXTENDEDCAPS was negotiated).

Responding to Inquiries

Sources must be able to respond to capability inquiries with current values at any time the Source is open (i.e. from MSG_OPENDS until MSG_CLOSEDS or before posting a MSG_CLOSEDSREQ).

A Source should respond with information to any capability that applies to your device. Only if a capability has no match with your device's features should you respond with TWRC_FAILURE / TWCC_BADCAP.

Refer to Chapter 9 for the complete list of TWAIN-defined capabilities.

Responding to Requests to Set Capabilities

If the requested value is invalid or the Source does not support the capability, then return TWRC_FAILURE / TWCC_CAPUNSUPPORTED. If the requested operation (MSG_SET, MSG_RESET, etc.) is not supported, then return TWRC_FAILURE / TWCC_CAPBADOPERATION. If the capability is unavailable because of a dependency on another capability (i.e., ICAP_CCITTKFACTOR is not available unless ICAP_COMPRESSION is TWCP_GROUP32D), then return TWCC_CAPSEQERROR.

If the request was fulfilled, return TWRC_SUCCESS.

If the requested value is close to an acceptable value but doesn't match exactly, set it as closely as possible and then return TWRC_CHECKSTATUS.

Memory Allocation

The TW_CAPABILITY structure used in capability negotiation is both allocated and deallocated by the application. The Container structure pointed to by the hContainer field in TW_CAPABILITY is allocated by the Source for “get” operations (MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, MSG_RESET) and by the application for the MSG_SET operation. Regardless of which one allocates the container, the application is responsible for deallocating it when it is done with it.

Limitations Imposed by the Negotiation

If a Source agrees to allow an application to restrict a capability, it is critical that the Source abide by that agreement. If at all possible, the Source’s user interface should reflect the agreement and not offer invalid options. The Source should never transfer data that violates the agreement reached during capability negotiation. In that situation, the Source can decide to fail the transfer or somehow adjust the values.

Data Transfers

Transfer Modes

All Sources must support Native and Buffered Memory data transfers. It is strongly suggested that they support Disk File mode, too. The default mode is Native. To select one of the other modes, the application must negotiate the ICAP_XFERMECH capability (whose default is TWSX_NATIVE). Sources must support negotiation of this capability. The native format for Microsoft Windows is DIB. For Macintosh, the native format is a PICT. The version of PICT to be transferred is the latest version available on the machine on which the application is running (usually PICT II for machines running 32-bit/color QuickDraw and PICT I for machines running black and white QuickDraw).

Initiating a Transfer

Transfers are initiated by the application (using the DG_IMAGE / DAT_IMAGExxxxFER / MSG_GET operations). A successful transfer transitions the session to State 7. If the transfer fails, the Source returns TWRC_FAILURE with the appropriate Condition Code and remains in State 6.

Concluding a Successful Transfer

To signal that the transfer is complete (i.e. the file is completed or the last buffer filled), the Source should return TWRC_XFERDONE. In response, the application must send a DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation. Only then may the Source transition from State 7 back to State 6 or to State 5 if no more images are ready to be transferred.

If more images are pending transfer, the Source must wait in State 6 until the application either requests the transfer or aborts the transfers. The Source may not “time-out” on any TWAIN transaction.

Aborting a Transfer

Either the application or Source can originate the termination of a transfer (although the application cannot do this in the middle of a Native or Disk File mode transfer). The Source generally terminates the transfer if the user cancels the transfer or a device error occurs which the Source determines is fatal to the transfer or the connection with the application. If the user canceled the transfer, the Source should return TWRC_CANCEL to signal the premature termination. The session remains in State 7 until the application sends the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation. If the Source aborts the transfer, it returns TWRC_FAILURE and the session typically remains in State 6. (If the failure occurs during the second buffer, or a later buffer, of a Buffered Memory transfer, the session remains in State 7.)

Native Mode Transfers

On Native mode transfers, the data parameter in the DSM_Entry call is a pointer to a variable of type TW_UINT32.

On Windows

The low word of this 32-bit integer is a handle variable to a DIB (Device Independent Bitmap) located in memory.

On Macintosh

This 32-bit integer is a handle to a Picture (a PicHandle). It is a Quick Draw picture located in memory.

Native transfers require the data to be transferred to a single large block of RAM. Therefore, they always face the risk of having an inadequate amount of RAM available to perform the transfer successfully.

If inadequate memory prevents the transfer, the Source has these options:

- Fail the transfer operation- Return TWRC_FAILURE / TWCC_LOWMEMORY
- Allow the user to clip the data to fit into available memory - Return TWRC_XFERDONE
- Allow the user to cancel the operation - Return TWRC_CANCEL

If the operation fails, the session remains in State 6. If the operation is canceled, the session remains in State 7 awaiting the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER or MSG_RESET from the application. The application can return the session to State 4 and attempt to renegotiate the transfer mechanism (ICAP_XFERMECH) to Disk File or Buffered Memory mode.

The Source cannot be interrupted by the application when it is acquiring an image through Native Mode Transfer. The Source's user interface may allow the user to abort the transfer, but the application will not be able to do so even if the application presents its own acquisition user interface.

Disk File Mode Transfers

The Source selects a default file format and file name (typically, TWAIN.TMP in the current directory). It reports this information to the application in response to the DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET operation.

The application may determine all of the Source's supported file formats by using the ICAP_IMAGEFILEFORMAT capability. Based on this information, the application can request a particular file format and define its own choice of file name for the transfer. The desired file format and file name will be communicated to the Source in a DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation.

When the Source receives the DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET operation, it should transfer the data into the designated file. The following conditions may exist:

Condition	**How to Handle**
No file name and/or file format was specified by the application during setup	Use either the Source's default file name or the last file information given to the Source by the application. Create the file.
The application specified a file but failed to create it	Create the application's defined file.
The application's specified file exists but has data in it	Overwrite the existing data.

The Source cannot be interrupted by the application when it is acquiring a file. The Source's user interface may allow the user to abort the transfer, but the application will not be able to do so even if the application presents its own acquisition user interface.

Buffered Memory Mode Transfers

When the Source transfers strips of data, the application allocates and deallocates buffers used for a Buffered Memory mode transfer. However, the Source must recommend appropriate sizes for those buffers and should check that the application has followed its recommendations.

When the Source transfers tiles of data, the Source allocates the buffers. The application is responsible for deallocating the memory.

To determine the Source's recommendations for buffer sizes, the application performs a DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET operation. The Source fills in the MinBufSize, MaxBufSize, and Preferred fields to communicate its buffer recommendations. Buffers must be double-word aligned and padded with zeros per raster line.

When an application issues a DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET operation, check the TW_IMAGEMEMXFER.Memory.Length field to determine the size of the buffer being presented to you. If it does not fit the recommendations, fail the operation with TWRC_FAILURE / TWCC_BADVALUE.

If the buffer is an appropriate size, fill in the required information.

- Sources must write one or more complete lines of image data (the full width of a strip or tile) into the buffer. Partial lines of image data are not allowed. If some of the buffer is unused, fill it in with zeros. Additionally, each line must be aligned to a 32-bit boundary. Return TWRC_SUCCESS after each successful buffer except for the last one (return TWRC_XFERDONE after that one).
- If the Source is transferring data whose bit depth is not 8 bits, it should fill the buffer without padding the data. If a 5-bit device wants the application to interpret its data as 8-bit data, it should report that it is supplying 8-bit data, make the valid data bits the most significant bits in the data byte, and pad the least significant bits with bits of whichever sense is "lightest". Otherwise, the Source should pack the data bits together. For a 5-bit R-G-B device, that means the data for the green channel should immediately follow the last bit of the red channel. The application is responsible for "unpacking" the data. The Source reports how many bits it is providing per pixel in the BitsPerPixel field of the TW_IMAGEINFO data structure.

Error Handling

Operation Triplet and State Verification

- Sources support all defined TWAIN triplets. A Source must verify every operation triplet they receive. If the operation is not recognized, the Source should return TWRC_FAILURE and TWCC_BADPROTOCOL.
- Sources must also maintain an awareness of what state their session is in. If an application invokes an operation that is invalid in the current state, the Source should fail the operation and return TWRC_FAILURE and TWCC_SEQERROR. Valid states for each operation are listed in Chapter 7.
- Anytime a Source fails an operation that would normally cause the session to transition to another state, the session should not transition but should remain in the original state.
- Each triplet operation has its own set of valid Return and Condition Codes as listed in Chapter 7. The Source must return a valid Return Code and set a valid Condition Code, if applicable, following every operation.
- All Return Codes and Condition Codes in the Source should be cleared upon the next call to DS_Entry( ). Clearing is delayed when a DG_CONTROL / DAT_STATUS / MSG_GET operation is received. In this case, the Source will fill the TW_STATUS structure with the current condition information and then clear that information.
- If an application attempts to connect to a Source that only supports single connection (or a multiply-connected Source that can't establish any new connections), the Source should respond with TWRC_FAILURE and TWCC_MAXCONNECTIONS.

- For Windows Sources only, the DLL implementation makes it possible to be connected to more than one application. Unless the operation request is to open the Source, the Source must verify the application originating an operation is currently connected to the Source. To do this:
  - ✓ The Source must maintain a list containing the Id value for each connected application. (The Id value comes from the application's TW_IDENTITY structure which is referenced by the pOrigin parameter in the DS_Entry( ) call.)
  - ✓ The Source should check the TW_IDENTITY.Id information of the application sending the operation and verify that it appears in the Source's list of connected applications.
- For Windows only, if connected to multiple applications, the Source is responsible for maintaining a separate, current Condition Code for each application it is connected to. The Source writer should also maintain a temporary, and separate, Condition Code for any application that is attempting to establish a connection with the Source. This is true both for Sources that support only a single connection or have reached the maximum connections.

Unrecoverable Error Situations

The Source is solely responsible for determining whether an error condition within the Source is recoverable or not. The Source must determine when, and what, error condition information to present to the user. The application relies on the Source to specify when a failure occurs. If a Source is in an unrecoverable error situation, it may send a MSG_CLOSEDSREQ to the application to request to have its user interface disabled and have an opportunity to begin again.

DAT_EVENT Handling Errors

One of the most common problems between a data source and application is the management of DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT. The symptoms are not immediately obvious, so it is worth mentioning them to assist new developers in quickly identifying and solving the problem.

Cannot use TAB or Keyboard Shortcuts to Navigate TWAIN Dialog

The cause of this can be one of two things. Either the application is not forwarding all messages to TWAIN through the DAT_EVENT mechanism, or the data source is not properly processing the DAT_EVENT messages. (Windows: calling IsDialogMessage for each forwarded message with TWAIN Dialog handle)

TWAIN Dialog Box Combo Boxes cannot be opened, Edit boxes produce multiple chars per keystroke

This case is caused by processing TWAIN Dialog Messages twice. Either the data source has not returned the proper return code in response to DAT_EVENT calls (Windows: TWRC_DSEVENT when IsDialogMessage returns TRUE), or the application is ignoring the return code.

This is not a problem when data source operates through TWAIN Thunker

Problems with the application handling of these messages are not often detected if the data source is operating through the TWAIN Thunking mechanism. This is because the Thunker process has a separate Window and Message pump that properly dispatch DAT_EVENT messages to the data source. Any mistake in application handling will pass without notice since all DAT_EVENT calls will return TWRC_NOTDSEVENT. (with the exception of important messages such as MSG_XFERREADY.)

Problem seems erratic, keyboard shortcuts and Tab key work for Message Boxes, but not TWAIN Dialog

This observation often further confuses the issue. In Windows, a standard Message box is Modal, and operates from a local message pump until the user closes it. All messages are properly dispatched to the message box since it does not rely on the application message pump. The TWAIN Dialog is slightly different since it is implemented Modeless. There is no easy way to duplicate Modal behavior for the TWAIN Dialog.

Memory Management

Windows Specifics

A single copy of the Source Manager and Source(s) services all applications wishing to access TWAIN functionality. If the Source can connect to more than one application, it will probably need to maintain a separate execution frame for each connected application. The Source does not have unlimited memory available to it so be conservative in its use.

It is valid for an application to open a Source and leave it open between several acquires. Therefore, Sources should minimize the time and resources required to load and remain open (in State 4). Also, Sources should allow a reasonable number of connections to ensure they can handle more than one application using the Source in this manner (leaving it open between acquires).

Macintosh Specifics

Each application that loads the Source Manager has a private copy of the Source Manager running within that application's heap space. Each Source that is connected runs as a private copy within the application's heap. It is important for the Source writer to recognize that their Source will be running in the memory frame of the host application, not in its own frame. Therefore, the Source should be conscientious with allocation and deallocation of memory.

General Guidelines

The following are some general guidelines:

- Check, when the Source is launched, to assure that enough memory space is available for adequate execution.
- Always verify that allocations were successful.
- Work with relocatable objects whenever possible - the heap you fragment is not your own.
- Deallocate temporary memory objects as soon as they are no longer needed.
- Maintain as small a non-operating memory footprint as can prudently be done - the Source will be "compatible" with more applications on more machines.
- Clean up after yourself. When about to be closed, deallocate all locally allocated RAM, eliminate any other objects on the heap, and prepare as appropriate to terminate.

Local Variables

The Source may allocate and maintain local variables and buffers. Remember that you are borrowing RAM from the application so be efficient about how much RAM is allocated simultaneously.

Instances Where the Source Allocates Memory

In general, the application allocates all necessary structures and passes them to the Source. There are a few exceptions to this rule:

- The Source must create the container, pointed to by the hContainer field, needed to hold capability information on DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, or MSG_RESET operations. The application deallocates the container.
- The Source allocates the buffer for Native mode data transfers. The application deallocates the buffer.
- Normally, the application creates the buffers used in a Buffered Memory transfer (DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET). However, if the Source is transferring tiled data, rather than strips of data, it is responsible for allocating the buffers. The application deallocates the buffers.

See the DG_IMAGE / DAT_JPEGCOMPRESSION operations.

Requirements to be a TWAIN-Compliant Source

Requirements

TWAIN-compliant Sources **must** support the following:

Operations

DG_CONTROL / DAT_CAPABILITY / MSG_GET
DG_CONTROL / DAT_CAPABILITY / MSG_GETCURRENT
DG_CONTROL / DAT_CAPABILITY / MSG_GETDEFAULT
DG_CONTROL / DAT_CAPABILITY / MSG_RESET
DG_CONTROL / DAT_CAPABILITY / MSG_SET

DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT

DG_CONTROL / DAT_IDENTITY / MSG_GET
DG_CONTROL / DAT_IDENTITY / MSG_OPENDS
DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS

DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER
DG_CONTROL / DAT_PENDINGXFERS / MSG_GET
DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET

DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET

DG_CONTROL / DAT_STATUS / MSG_GET

DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS
DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

DG_CONTROL / DAT_XFERGROUP / MSG_GET

DG_IMAGE / DAT_IMAGEINFO / MSG_GET

DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT
DG_IMAGE / DAT_IMAGELAYOUT / MSG_RESET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET

DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET

DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET

Capabilities

Every Source must support all five DG_CONTROL / DAT_CAPABILITY operations on:

CAP_XFERCOUNT

Every Source must support DG_CONTROL / DAT_CAPABILITY MSG_GET on:

CAP_SUPPORTEDCAPS
CAP_UICONTROLLABLE

Sources that supply image information must support DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULTon:

ICAP_COMPRESSION
ICAP_PLANARCHUNKY
ICAP_PHYSICALHEIGHT
ICAP_PHYSICALWIDTH
ICAP_PIXELFLAVOR

Sources that supply image information must support DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, MSG_RESET and MSG_SET on:

ICAP_BITDEPTH
ICAP_BITORDER
ICAP_PIXELTYPE
ICAP_UNITS
ICAP_XFERMECH
ICAP_XRESOLUTION
ICAP_YRESOLUTION

All Sources must implement the advertised features supported by their devices. They must make these features available to applications via the TWAIN protocol. For example, a Source that's connected to a device that has an ADF must support DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT on:

CAP_FEEDERENABLED
CAP_FEEDERLOADED

and DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, MSG_RESET and MSG_SET on:

CAP_AUTOFEED

If the ADF also supports ejecting and rewinding of pages then the Source should also support DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, MSG_RESET and MSG_SET on:

CAP_CLEARPAGE
CAP_REWINDPAGE

Other Topics

Custom Operations

Manufacturers may add custom operations to their Sources. These can also be made known to application manufacturers. This mechanism allows an application to access functionality not normally available from a generic TWAIN Source.

One use of this mechanism might be to implement device-specific diagnostics for a hardware diagnostic program. These custom operations should be used sparingly and never in place of pre-defined TWAIN operations.

Custom operations are defined by specifying special values for Data Groups (DGs), Data Argument Types (DATs), Messages (MSGs), and Capabilities (CAPs). The following areas have been reserved for custom definitions:

Data Groups	**Top 8 bit flags (bits 24 - 31) in the DG identifiers reserved for custom use.**
DATs	Designators with values greater than 8000 hex.
Messages	Designators with values greater than 8000 hex.
Capabilities	Designators with values greater than 8000 hex.

The responsibility for naming and managing the use of custom designators lies wholly upon the TWAIN element originating the designator and the element consuming it. Prior to interpreting a custom designator, the consuming element must check the originating element’s ProductName string from its TW_IDENTITY structure. Since custom operation numbers may overlap, this is the only way to insure against confusion.

Networking

If a Source supports connection to a remote device over a network, the Source is responsible for hiding the network dependencies of that device's operation from the application. The Source Manager does not know anything about networks.

In a networking situation, the Source will probably be built in two segments: One running on the machine local to the application, the other running remotely across the network. Sources are required to handle all the network interfacing with remote devices (real or logical) through local Source "stubs" that understand how to access both the network and the remote Source while interacting logically with the Source Manager.

The segment running on the local machine will probably be a "stub" Source. That is, the local stub will translate all operations received from the application and Source Manager into a form the remote source understands (that is, not necessarily TWAIN-defined operations). The stub also:

- Converts the information returned from the remote source into TWAIN-compliant results
- Handles local memory management for data copies and data transferring
- Isolates the network from the Source Manager and application
- Manages the connection with the remote Source
- Provides any needed code to handle local hardware (such as interface hardware)
- Provides the local user interface to control the remote Source

6

Entry Points and Triplet Components

Chapter Contents

Entry Points

TWAIN has two entry points:

- **DSM_Entry( ) -** located in the Source Manager and typically called by applications, with the following exceptions where a Windows Source calls the Source Manager to communicate with an Application:

  DG_CONTROL / DAT_NULL / MSG_XFERREADY
  DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ
  DG_CONTROL / DAT_NULL / MSG_CLOSEDSOK
  DG_CONTROL / DAT_NULL / MSG_DEVICEEVENT

- **DS_Entry( ) -** located in the Source and called only by the Source Manager.

Programming Basics

- Upon entry, the parameters must be ordered on the stack in Pascal form. Be sure that your code expects this ordering rather than the reverse order that C uses.
- The keyword FAR is included in the entry point syntax to accommodate the Windows/DOS segmented addressing scheme. It has no value for Macintosh or UNIX systems so the TWAIN.H file simply defines FAR as an empty value for Macintosh and UNIX.

Declaration of DSM_Entry( )

Written in C code form, the declaration looks like this:

On Windows

```
TW_UINT16 FAR PASCAL DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       pTW_IDENTITY   pDest,      // destination of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

On Macintosh

```
FAR PASCAL TW_UINT16 DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       pTW_IDENTITY   pDest,      // destination of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

Parameters of DSM_Entry( )

pOrigin

This points to a TW_IDENTITY structure, allocated by the application, that describes the application making the call. One of the fields in this structure, called Id, is an arbitrary and unique identifier assigned by the Source Manager to tag the application as a unique TWAIN entity. The Source Manager maintains a copy of the application's identity structure, so the application must not modify that structure unless it first breaks its connection with the Source Manager, then reconnects to cause the Source Manager to store the new, modified identity.

pDest

This is set either to NULL if the application is aiming the operation at the Source Manager or to the TW_IDENTITY structure of the Source that the application is attempting to reach. The application allocated the space for the Source's identity structure when it decided which Source was to be connected. The Source's TW_IDENTITY.Id is also uniquely set by the Source Manager when the Source is opened and should not be modified by the Source. The application should not count on the value of this field being consistent from one session to the next because the Source Manager reallocates these numbers every time it is opened. The Source Manager keeps a copy of the Source's identity structure as should the application and the Source.

DG

The Data Group of the operation triplet. Currently, only DG_CONTROL, DG_IMAGE, and DG_AUDIO are defined.

DAT

The Data Argument Type of the operation triplet. A complete list appears later in this chapter.

MSG

The Message of the operation triplet. A complete list appears later in this chapter.

pData

The pData parameter is of type TW_MEMREF and is a pointer to the data (a variable or, more typically, a structure) that will be used according to the action specified by the operation triplet.

Declaration of DS_Entry( )

DS_Entry is only called by the Source Manager. Written in C code form, the declaration looks like this:

On Windows

```
TW_UINT16 FAR PASCAL DS_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

On Macintosh

```
FAR PASCAL TW_UINT16 DS_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

Data Groups

TWAIN operations can be broadly classified into three data groups:

Control Oriented (DG_CONTROL)

Controls the TWAIN session. Consumed by both Source Manager and Source. It is always available, no matter what the current setting of DG_CONTROL / DAT_XFERGROUP.

Image Data Oriented (DG_IMAGE)

Indicates the kind of data to be transferred. Change between DG_AUDIO and DG_IMAGE as needed using DG_CONTROL / DAT_XFERGROUP / MSG_SET. The default at startup is for a Source to be ready to transfer DG_IMAGE data.

Audio Data Oriented (DG_AUDIO)

Indicates the kind of data to be transferred. Change between DG_AUDIO and DG_IMAGE as needed using DG_CONTROL / DAT_XFERGROUP / MSG_SET.

Currently, only image and audio data are supported but this could be expanded to include text, etc. This has several future implications. If more than one data type exists, an application and a Source will need to decide what type(s) of data the Source can, and will be allowed to, produce before a transfer can occur. Further, if multiple transfers are being generated from a single acquisition—such as when image and text are intermixed and captured from the same page—it must be unambiguous which type of data is being returned from each data transfer.

Programming Basics

Note the following:

- Data Group designators are 32-bit, unsigned values. The actual values that are assigned are powers of two (bit flags) so that the DGs can be easily masked.
- There are 24 DGs designated as “reserved” for pre-defined DGs . Four are currently in use. The top 8 bits are reserved for custom DGs.

Data Argument Types

Data Argument Types, or DATs, are used to allow programmatic identification of the TWAIN type for the structure of status variable referenced by the entry point parameter pData. pData will always point to a variable or data structure defined by TWAIN. If the consuming application or Source switches (cases, etc.) on the DAT specified in the formal parameter list of the entry point call, it can handle the form of the referenced data correctly.

Table 6-1. Data Argument Types

Data Type	Used by	Associated structure or type
DAT_NULL	ANY DG	Null structure. No data required for the operation
DAT_CUSTOMBASE	n/a	Not a DAT in itself, but the baseline a Source must use when creating a custom DAT.
DAT_AUDIOFILEXFER	DG_AUDIO	Operates on null data. Filename / Format already negotiated.
DAT_AUDIONATIVEXFER	DG_AUDIO	TW_UINT32
		On Windows - low word = WAV handle
		On Macintosh - audio handle
DAT_CAPABILITY	DG_CONTROL	TW_CAPABILITY structure
DAT_EVENT	DG_CONTROL	TW_EVENT structure
DAT_FILESYSTEM	DG_CONTROL	TW_FILESYSTEM structure
DAT_IDENTITY	DG_CONTROL	TW_IDENTITY structure
DAT_PARENT	DG_CONTROL	TW_INT32
		On Windows - low word=Window handle
		On Macintosh - Set to NULL
DAT_PASSTHRU	DG_CONTROL	TW_PASSTHRU structure
DAT_PENDINGXFERS	DG_CONTROL	TW_PENDINGXFERS structure
DAT_SETUPFILEXFER	DG_CONTROL	TW_SETUPFILEXFER structure
DAT_SETUPMEMXFER	DG_CONTROL	TW_SETUPMEMXFER structure
DAT_STATUS	DG_CONTROL	TW_STATUS structure
DAT_USERINTERFACE	DG_CONTROL	TW_USERINTERFACE structure
DAT_XFERGROUP	DG_CONTROL	TW_UINT32
		A DG designator describing data to be transferred (currently only image data is supported)
DAT_CIECOLOR	DG_IMAGE	TW_CIECOLOR structure
DAT_GRAYRESPONSE	DG_IMAGE	TW_GRAYRESPONSE structure

DAT_IMAGEFILEXFER	DG_IMAGE	Operates on NULL data. Filename/Format already negotiated
DAT_IMAGEINFO	DG_IMAGE	TW_IMAGEINFO structure
DAT_IMAGELAYOUT	DG_IMAGE	TW_IMAGELAYOUT structure
DAT_IMAGEMEMXFER	DG_IMAGE	TW_IMAGEMEMXFER structure
DAT_IMAGENATIVEXFER	DG_IMAGE	TW_UINT32;
		On Windows - low word=DIB handle
		On Macintosh - PicHandle
DAT_JPEGCOMPRESSION	DG_IMAGE	TW_JPEGCOMPRESSION structure
DAT_PALETTE8	DG_IMAGE	TW_PALETTE8 structure
DAT_RGBRESPONSE	DG_IMAGE	TW_RGBRESPONSE structure

Messages

A Message, or MSG, is used to communicate between TWAIN elements what action is to be taken upon a particular piece of data, or for a data-less operation, what action to perform. If an application wants to make anything happen in, or inquire any information from, a Source or the Source Manager, it must make a call to DSM_Entry( ) with the proper MSG as one parameter of the operation triplet. The data to be acted upon is also specified in the parameter list of this call.

A MSG is always associated with a Data Group (DG) identifier and a Data Argument Type (DAT) identifier in an operation triplet. This operation unambiguously specifies what action is to be taken on what data. Refer to Chapter 7 for the list of defined operation triplets.

Table 6-2. Messages

Message ID	Valid DAT(s)	Description of Specified Action
MSG_AUTOMATICCAPTURED DIRECTORY	DAT_FILESYSTEM	Place to store images acquired during automatic capture
MSG_CHANGEDIRECTORY	DAT_FILESYSTEM	Change device, domain, host, or image directory
MSG_CLOSEDS	DAT_IDENTITY	Close the specified Source
MSG_CLOSEDSM	DAT_PARENT	Close the Source Manager
MSG_CLOSEDSREQ	DAT_NULL	Source requests for application to close Source

MSG_COPY	DAT_FILESYSTEM	Copy images across storage devices
MSG_CREATEDIRECTORY	DAT_FILESYSTEM	Create an image directory
MSG_CUSTOMBASE	n/a	Not a message in itself, but the baseline a Source must use when creating a custom message
MSG_DELETE	DAT_FILESYSTEM	Delete an image or an image directory
MSG_DEVICEEVENT	DAT_NULL	Report an event from the Source to the Source Manager
MSG_DISABLEDS	DAT_USERINTERFACE	Disable data transfer in the Source
MSG_ENABLEDS	DAT_USERINTERFACE	Enable data transfer in the Source
MSG_ENDXFER	DAT_PENDINGXFERS	Application tells Source that transfer is over
MSG_FORMATMEDIA	DAT_FILESYSTEM	Format a storage device
MSG_GET	various DATs	Get all Available Values including Current & Default
MSG_GETCLOSE	DAT_FILESYSTEM	Close a file context created by MSG_GETFIRSTFILE
MSG_GETCURRENT	various DATs	Get Current value
MSG_GETDEFAULT	various DATs	Get Source's preferred default value
MSG_GETFIRST	DAT_IDENTITY	Get first element from a "list"
MSG_GETFIRSTFILE	DAT_FILESYSTEM	Get the first file in a directory
MSG_GETINFO	DAT_FILESYSTEM	Get information about the current file
MSG_GETNEXT	DAT_IDENTITY	Get next element from a "list"
MSG_GETNEXTFILE	DAT_FILESYSTEM	Get the next file in a directory
MSG_NULL	None	No action to be taken
MSG_OPENDS	DAT_IDENTITY	Open and Initialize the specified Source
MSG_OPENDSM	DAT_PARENT	Open the Source Manager
MSG_PASSTHRU	DAT_PASSTHRU	For use by Source Vendors only
MSG_PROCESSEVENT	DAT_EVENT	Tells Source to check if event/message belongs to it
MSG_RENAME	DAT_FILESYSTEM	Rename an image or an image directory
MSG_RESET	various DATs	Return specified item to power-on (TWAIN default) condition
MSG_SET	various DATs	Set one or more values
MSG_USERSELECT	DAT_IDENTITY	Presents dialog of all Sources to select from
MSG_XFERREADY	DAT_NULL	The Source has data ready for transfer to the application

Custom Components of Triplets

Custom Data Groups

A manufacturer may choose to implement custom data descriptors that require a new Data Group. This would be needed if someone decides to extend TWAIN to, say, satellite telemetry.

- The top 8 bits of every DG_xxxx identifier are reserved for use as custom DGs. Custom DG identifiers must use one of the upper 8 bits of the DG_xxxx identifier. Remember, DGs are bit flags.
- The originator of the custom DG must fill the ProductName field in the application or Source's TW_IDENTITY structure with a uniquely descriptive name. The consumer will look at this field to determine whose custom DG is being used.
- TWAIN provides no formal allocation (or vendor-specific "identifier blocks") for custom data group identifiers nor does it do any coordination to avoid collisions.
- The DG_CUSTOMBASE value resides in the TWAIN.H file. All custom IDs must be numerically greater than this base. A similar custom base "address" is defined for Data Argument Types, Messages, Capabilities, Return Codes, and Condition Codes. The only difference in concept is that DGs are the only designators defined as bit flags. All other custom values can be any integer value larger than the xxxx_CUSTOMBASE defined for that type of designator.

Custom Data Argument Types

DAT_CUSTOMBASE is defined in the TWAIN.H file to allow a Source vendor to define "custom" DATs for their particular device(s). The application can recognize the Source by checking the TW_IDENTITY.ProductName and the TW_IDENTITY.TW_VERSION structure. If an application is aware that this particular Source offers custom DATs, it can use them. No changes to TWAIN or the Source Manager are required to support such identifiers (or the data structures which they imply).

Refer to the TWAIN.H file for the value of DAT_CUSTOMBASE for custom DATs. All custom values must be numerically greater than this constant.

Custom Messages

As with the DATs, MSG_CUSTOMBASE is included in TWAIN.H so that the Source writer can create custom messages specific to their Source. If the applications understand these custom messages, actions beyond those defined in this specification can be performed through the normal TWAIN mechanism. No modifications to TWAIN or the Source Manager are required.

Remember that the consumer of these custom values will look in your TW_IDENTITY.ProductName field to clarify what the identifier's value means—there is no other protection for overlapping custom definitions. Refer to the TWAIN.H file for the value of MSG_CUSTOMBASE for custom Messages. All custom values must be numerically greater than this value.

7

Operation Triplets

Chapter Contents

An Overview of the Triplets

From Application to Source Manager (Control Information)

Data Group	Data Argument Type	Message	Page #
DG_CONTROL	DAT_IDENTITY	MSG_CLOSEDS	7-174
		MSG_GETDEFAULT	7-177
		MSG_GETFIRST	7-178
		MSG_GETNEXT	7-180
		MSG_OPENDS	7-182
		MSG_USERSELECT	7-186
DG_CONTROL	DAT_PARENT	MSG_CLOSEDSM	7-193
		MSG_OPENDSM	7-194
DG_CONTROL	DAT_STATUS	MSG_GET	7-208

From Application to Source (Control Information)

Data Group	Data Argument Type	Message	Page #
DG_CONTROL	DAT_CAPABILITY	MSG_GET	7-145
		MSG_GETCURRENT	7-147
		MSG_GETDEFAULT	7-149
		MSG_QUERYSUPPORT	7-151
		MSG_RESET	7-153
		MSG_SET	7-155
DG_CONTROL	DAT_CUSTOMDSDATA	MSG_GET	7-158
		MSG_SET	7-159
DG_CONTROL	DAT_FILESYSTEM	MSG_CHANGEDIRECTORY	7-163
		MSG_COPY	7-165
		MSG_CREATEDIRECTORY	7-166
		MSG_DELETE	7-167
		MSG_FORMATMEDIA	7-168
		MSG_GETCLOSE	7-169
		MSG_GETFIRSTFILE	7-170
		MSG_GETINFO	7-171
		MSG_GETNEXTFILE	7-172
		MSG_RENAME	7-173
DG_CONTROL	DAT_EVENT	MSG_PROCESSEVENT	7-161
DG_CONTROL	DAT_PASSTHRU	MSG_PASSTHRU	7-195
DG_CONTROL	DAT_PENDINGXFERS	MSG_ENDXFER	7-196
		MSG_GET	7-198
		MSG_RESET	7-200
DG_CONTROL	DAT_SETUPFILEXFER	MSG_GET	7-202
		MSG_GETDEFAULT	7-203
		MSG_RESET	7-204
		MSG_SET	7-205
DG_CONTROL	DAT_SETUPMEMXFER	MSG_GET	7-207
DG_CONTROL	DAT_STATUS	MSG_GET	7-208
DG_CONTROL	DAT_USERINTERFACE	MSG_DISABLEDS	7-210
		MSG_ENABLEDS	7-211
		MSG_ENABLEDSUIONLY	7-214
DG_CONTROL	DAT_XFERGROUP	MSG_GET	7-215

From Application to Source (Image Information)

Data Group	Data Argument Type	Message	Page #
DG_IMAGE	DAT_CIECOLOR	MSG_GET	7-216
DG_IMAGE	DAT_EXTIMAGEINFO	MSG_GET	7-218
DG_IMAGE	DAT_GRAYRESPONSE	MSG_RESET	7-220
		MSG_SET	7-221
DG_IMAGE	DAT_IMAGEFILEXFER	MSG_GET	7-222
DG_IMAGE	DAT_IMAGEINFO	MSG_GET	7-224
DG_IMAGE	DAT_IMAGELAYOUT	MSG_GET	7-226
		MSG_GETDEFAULT	7-228
		MSG_RESET	7-229
		MSG_SET	7-230
DG_IMAGE	DAT_IMAGEMEMXFER	MSG_GET	7-232
DG_IMAGE	DAT_IMAGENATIVEXFER	MSG_GET	7-234
DG_IMAGE	DAT_JPEGCOMPRESSION	MSG_GET	7-237
		MSG_GETDEFAULT	7-238
		MSG_RESET	7-239
		MSG_SET	7-240
DG_IMAGE	DAT_PALETTE8	MSG_GET	7-241
		MSG_GETDEFAULT	7-243
		MSG_RESET	7-244
		MSG_SET	7-245
DG_IMAGE	DAT_RGBRESPONSE	MSG_RESET	7-246
		MSG_SET	7-248

From Application to Source (Audio Information)

Data Group	Data Argument Type	Message	Page #
DG_AUDIO	DAT_AUDIOFILEXFER	MSG_GET	7-142
DG_AUDIO	DAT_AUDIOINFO	MSG_GET	7-143
DG_AUDIO	DAT_AUDIONATIVEXFER	MSG_GET	7-144

From Source Manager to Source (Control Information)

Data Group	Data Argument Type	Message	Page #
DG_CONTROL	DAT_IDENTITY	MSG_CLOSEDS	7-174
		MSG_GET	7-176
		MSG_OPENDS	7-184

From Source to Application (Control Information via the Source Manager)

(Used by Windows Sources only)

Data Group	Data Argument Type	Message	Page #
DG_CONTROL	DAT_NULL	MSG_CLOSEDSOK	
		MSG_CLOSEDSREQ	7-188
		MSG_DEVICEEVENT	7-190
		MSG_XFERREADY	7-191

Format of the Operation Triplet Descriptions

The following pages describe the operation triplets. They are all included and are arranged in alphabetical order using the Data Group, Data Argument Type, and Message identifier list.

There are three operations that are duplicated because that have a different originator and/or destination in each case. They are:

- DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS
  - ✓ from Application to Source Manager
  - ✓ from Source Manager to Source
- DG_CONTROL / DAT_IDENTITY / MSG_OPENDS
  - ✓ from Application to Source Manager
  - ✓ from Source Manager to Source
- DG_CONTROL / DAT_STATUS / MSG_GET
  - ✓ from Application to Source Manager
  - ✓ from Application to Source

The format of each page is:

Triplet - The Concise DG / DAT / MSG Information

Call

Actual format of the routine call (parameter list) for the operation. Identification of the data structure used for the pData parameter is included.

Valid States

The states in which the application, Source Manager, or Source may legally invoke the operation.

Description

General description of the operation.

Origin of the Operation (Application, Source Manager or, Source)

The action(s) the application, Source Manager, or Source should take before invoking the operation.

Destination of the Operation (Source Manager or Source)

The action that the destination element (Source Manager or Source) of the operation will take.

Return Codes

The Return Codes and Condition Codes that are defined and valid for this operation.

See Also

Lists other related operation triplets, capabilities, constants, etc.

Operation Triplets

DG_AUDIO / DAT_AUDIOFILEXFER / MSG_GET

Call

```
DSM_Entry (pOrigin, pDest, DG_AUDIO, DAT_AUDIOFILEXFER, MSG_GET, NULL);
```

Valid States

6 (transitions to state 7)

Description

(Similar operation to DAT_IMAGEFILEXFER).

This operation is used to initiate the transfer of audio from the Source to the application via the disk-file transfer mechanism. It causes the transfer to begin.

No special set up or action required. Application should have already invoked the DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation unless the Source's default transfer format and file name (typically, TWAINAUD.TMP) are acceptable to the application. The application need only invoke this operation once per image transferred.

Source should acquire the audio data, format it, create any appropriate header information, and write everything into the file specified by the previous DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation, and close the file.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL.
   TWCC_SEQERROR - not state 6.
```

See Also

ACAP_XFERMECH

DG_AUDIO / DAT_AUDIOINFO / MSG_GET

Call

```
DSM_Entry (pOrigin, pDest, DG_AUDIO, DAT_AUDIOINFO, MSG_GET,
    pSourceAudioInfo);
```

pSourceAudioInfo = A pointer to a TW_AUDIOINFO structure

Valid States

6 and 7

Description

Used to get the information of the current audio data ready to transfer. (Similar operation to DAT_IMAGEINFO)

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
    TWCC_BADPROTOCOL
    TWCC_SEQERROR
```

See Also

DG_AUDIO / DAT_AUDIONATIVEXFER / MSG_GET

Call

```
DSM_Entry (pOrigin, pDest, DG_AUDIO, DAT_AUDIONATIVEXFER, MSG_GET, pHandle);
```

pHandle = A pointer to a variable of type TW_UINT32

On Windows - This 32 bit integer is a handle variable to WAV data located in memory.

On Macintosh - This 32-bit integer is a handle to AIFF data

Valid States

6 (transitions to state 7)

Description

(Similar operation to DAT_IMAGENATIVEXFER).

Causes the transfer of an audioÆs data from the Source to the application, via the Native transfer mechanism, to begin. The resulting data is stored in main memory in a single block. The data is stored in AIFF format on the Macintosh and as a WAV format under Microsoft Windows. The size of the audio snippet that can be transferred is limited to the size of the memory block that can be allocated by the Source.

Note: This is the default transfer mechanism. All Sources support this mechanism if DG_AUDIO is supported. The Source will use this mechanism unless the application explicitly negotiates a different transfer mechanism with ACAP_XFERMECH.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
    TWCC_BADPROTOCOL.
    TWCC_SEQERROR - not state 6.
```

See Also

ACAP_XFERMECH

DG_CONTROL / DAT_CAPABILITY / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CAPABILITY, MSG_GET, pCapability);
```

pCapability = A pointer to a TW_CAPABILITY structure.

Valid States

4 through 7

Description

Returns the Source’s Current, Default and Available Values for a specified capability.

These values reflect previous MSG_SET operations on the capability, or Source’s automatic changes. (See MSG_SET).

Note: This operation does not change the Current or Available Values of the capability.

Application

Set the pCapability fields as follows:

```
pCapability->Cap = the CAP_xxxx or ACAP_xxxx or ICAP_xxxx identifier
pCapability->ConType = TWON_DONTCARE16
pCapability->hContainer = NULL
```

The Source will allocate the memory for the necessary container structure but the application must free it when the operation is complete and the application no longer needs to maintain the information.

Use MSG_GET:

- As the first step in negotiation of a capability’s Available Values.
- To check the results if a MSG_SET returns TWRC_CHECKSTATUS.
- To check the Available, Current and Default Values with one command.

This operation may fail for a low memory condition. Either recover from a TWCC_LOWMEMORY failure by freeing memory for the Source to use so it can continue, or terminating the acquisition and notifying the user of the low memory problem.

Source

If the application requests this operation on a capability your Source does not recognize (and you’re sure you’ve implemented all the capabilities that you’re required to), disregard the operation, but return TWRC_FAILURE with TWCC_BADCAP.

If you support the capability, fill in the fields listed below and allocate the container structure and place its handle into pCapability->hContainer. The container should be referenced by a “handle” of type TW_HANDLE.

Fill the fields in pCapability as follows:

```
pCapability->ConType = TWON_ARRAY,
TWON_ONEVALUE,
TWON_ENUMERATION, or
TWON_RANGE

pCapability->hContainer = TW_HANDLE referencing a container of ConType
```

Set ConType to the container type your Source uses for this capability. For container types of TWON_ARRAY and TWON_ONEVALUE provide the Current Value. For container types TWON_ENUMERATION and TWON_RANGE provide the Current, Default and Available Values.

This is a memory allocation operation. It is possible for this operation to fail due to a low memory condition. Be sure to verify that the allocation is successful. If it is not, attempt to reduce the amount of memory occupied by the Source. If the allocation cannot be made, return TWRC_FAILURE with TWCC_LOWMEMORY to the application and set the pCapability->hContainer handle to NULL.

Note that the Source **must** be able to respond to an inquiry about any of its capabilities at **any time** that the Source is open.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
  TWCC_BADCAP        /* Unknown capability--Source does not */
                     /* does not recognize this capability  */
  TWCC_BADDEST       /* No such Source in session with      */
                     /* application                         */
  TWCC_LOWMEMORY     /* Not enough memory to complete the   */
                     /* operation                           */
  TWCC_SEQERROR      /* Operation invoked in invalid state  */
```

See Also

DG_CONTROL / DAT_CAPABILITY / MSG_GETCURRENT, MSG_GETDEFAULT, MSG_RESET, and MSG_SET

Capability Constants (in Chapter 8)

Capability Containers: TW_ONEVALUE, TW_ENUMERATION, TW_RANGE, TW_ARRAY (in Chapter 8)

Listing of all capabilities (in Chapter 9)

DG_CONTROL / DAT_CAPABILITY / MSG_GETCURRENT

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CAPABILITY, MSG_GETCURRENT,
pCapability);
```

pCapability = A pointer to a TW_CAPABILITY structure.

Valid States

4 through 7

Description

Returns the Source's Current Value for the specified capability.

The Current Value reflects previous MSG_SET operations on the capability, or Source's automatic changes. (See MSG_SET).

Note: This operation does not change the Current Values of the capability.

Application

Set the pCapability fields as follows:

```
pCapability->Cap = the CAP_xxxx or ACAP_xxxx or ICAP_xxxx identifier
pCapability->ConType = TWON_DONTCARE16
pCapability->hContainer = NULL
```

The Source will allocate the memory for the necessary container structure but the application must free it when the operation is complete and the application no longer needs to maintain the information.

Use MSG_GETCURRENT:

- To check the Source's power-on Current Values (see Chapter 9 for TWAIN-defined defaults for each capability).
- To check just the Current Value (in place of using MSG_GET).
- In State 6 to determine the settings. They could have been set by the user (if TW_USERINTERFACE.ShowUI = TRUE) or be the results of automatic processes used by the Source.

This operation may fail for a low memory condition. Either recover from a TWCC_LOWMEMORY failure by freeing memory for the Source to use so it can continue, or terminating the acquisition and notifying the user of the low memory problem.

Source

If the application requests this operation on a capability your Source does not recognize (and you're sure you've implemented all the capabilities that you're required to), disregard the operation, but return TWRC_FAILURE with TWCC_BADCAP.

If you support the capability, fill in the fields listed below and allocate the container structure and place its handle into pCapability->hContainer. The container should be referenced by a "handle" of type TW_HANDLE.

Fill the fields in pCapability as follows:

```
pCapability->ConType = TWON_ARRAY or TWON_ONEVALUE
pCapability->hContainer = TW_HANDLE referencing a container of ConType
```

Set ConType to the container type that matches the type for this capability. Fill the fields in the container structure with the Current Value of the capability.

This is a memory allocation operation. It is possible for this operation to fail due to a low memory condition. Be sure to verify that the allocation is successful. If it is not, attempt to reduce the amount of memory occupied by the Source. If the allocation cannot be made, return TWRC_FAILURE with TWCC_LOWMEMORY to the application and set the pCapability->hContainer handle to NULL.

Note that the Source **must** be able to respond to an inquiry about any of its capabilities at **any time** that the Source is open.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
  TWCC_BADCAP        /* unknown capability--Source does not  */
                     /* recognize this capability            */
  TWCC_BADDEST       /* No such Source in-session with       */
                     /* application                          */
  TWCC_LOWMEMORY     /* Not enough memory to complete the    */
                     /* operation                            */
  TWCC_SEQERROR      /* Operation invoked in invalid  state  */
```

See Also

DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETDEFAULT, MSG_RESET, and MSG_SET

Capability Constants (in Chapter 8)

Capability Containers: TW_ONEVALUE, TW_ENUMERATION, TW_RANGE, TW_ARRAY (in Chapter 8)

Listing of all capabilities (in Chapter 9)

DG_CONTROL / DAT_CAPABILITY / MSG_GETDEFAULT

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CAPABILITY, MSG_GETDEFAULT,
pCapability);
```

pCapability = A pointer to a TW_CAPABILITY structure.

Valid States

4 through 7

Description

Returns the Source's Default Value. This is the Source's preferred default value.

The Source's Default Value cannot be changed.

Application

Set the pCapability fields as follows:

```
   pCapability->Cap = the CAP_xxxx or ACAP_xxxx or ICAP_xxxx identifier
   pCapability->ConType = TWON_DONTCARE16
   pCapability->hContainer = NULL
```

The Source will allocate the memory for the necessary container structure but the application must free it when the operation is complete and the application no longer needs to maintain the information.

Use MSG_GETDEFAULT:

- To check the Source's preferred Values. Using the Source's preferred default as the Current Value may increase performance in some Sources.

This operation may fail for a low memory condition. Either recover from a TWCC_LOWMEMORY failure by freeing memory for the Source to use so it can continue, or terminating the acquisition and notifying the user of the low memory problem.

Source

If the application requests this operation on a capability your Source does not recognize (and you are sure you have implemented all the capabilities that you're required to), disregard the operation, but return TWRC_FAILURE with TWCC_BADCAP.

If you support the capability, fill in the fields listed below and allocate the container structure and place its handle into pCapability->hContainer. The container should be referenced by a "handle" of type TW_HANDLE.

Fill the fields in pCapability as follows:

```
pCapability->ConType = TWON_ARRAY or TWON_ONEVALUE
pCapability->hContainer = TW_HANDLE referencing a container of ConType
```

Set ConType to the container type that matches for this capability. Fill the fields in the container with the Default Value of this capability.

The Default Value is the preferred value for the Source. This value is used as the power-on value for capabilities if TWAIN does not specify a default.

This is a memory allocation operation. It is possible for this operation to fail due to a low memory condition. Be sure to verify that the allocation is successful. If it is not, attempt to reduce the amount of memory occupied by the Source. If the allocation cannot be made return TWRC_FAILURE with TWCC_LOWMEMORY to the application and set the pCapability->hContainer handle to NULL.

Note that the Source **must** be able to respond to an inquiry about any of its capabilities at **any time** that the Source is open.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
  TWCC_BADCAP         /* unknown capability--Source does not */
                      /* recognize this capability           */
  TWCC_BADDEST        /* No such Source in-session with      */
                      /* application                         */
  TWCC_LOWMEMORY      /* Not enough memory to complete the   */
                      /* operation                           */
  TWCC_SEQERROR       /* Operation invoked in invalid state  */
```

See Also

DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_RESET, and MSG_SET

Capability Constants (in Chapter 8)

Capability Containers: TW_ONEVALUE, TW_ENUMERATION, TW_RANGE, TW_ARRAY (in Chapter 8)

Listing of all capabilities (in Chapter 9)

DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CAPABILITY, MSG_GETDEFAULT,
pCapability);
```

pCapability = A pointer to a TW_CAPABILITY structure.

Valid States

4 through 7

Description

Returns the Source's support status of this capability.

Application

Set the pCapability fields as follows:

```
pCapability->Cap = the CAP_xxxx or ACAP_xxxx or ICAP_xxxx identifier
pCapability->ConType = TWON_ONEVALUE
pCapability->hContainer = NULL
```

The Source will allocate the memory for the necessary container structure but the application must free it when the operation is complete and the application no longer needs to maintain the information.

Use MSG_QUERYSUPPORT:

- To check the whether the Source supports a particular operation on the capability.

This operation may fail for a low memory condition. Either recover from a TWCC_LOWMEMORY failure by freeing memory for the Source to use so it can continue, or terminating the acquisition and notifying the user of the low memory problem.

Source

If the application requests this operation on a capability your Source does not recognize (and you're sure you've implemented all the capabilities that you're required to), do not disregard the operation, but fill out the TWON_ONEVALUE container with

Fill the fields in pCapability as follows:

```
pCapability->ConType = TWON_ONEVALUE

pCapability->hContainer = TW_HANDLE referencing a container of type
TW_ONEVALUE.
```

Fill the fields in TW_ONVALUE as follows:

ItemType =	TWTW_INT32;
Item =	Bit pattern representing the set of operation that are supported by the Data Source on this capability (TWQC_GET, TWQC_SET, TWQC_GETDEFAULT, TWQC_RESET);

This is a memory allocation operation. It is possible for this operation to fail due to a low memory condition. Be sure to verify that the allocation is successful. If it is not, attempt to reduce the amount of memory occupied by the Source. If the allocation cannot be made return TWRC_FAILURE with TWCC_LOWMEMORY to the application and set the pCapability->hContainer handle to NULL.

Note that the Source **must** be able to respond to an inquiry about any of its capabilities at **any time** that the Source is open.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
TWCC_BADDEST          /* No such Source in-session with      */
                      /* application                         */
TWCC_LOWMEMORY        /* Not enough memory to complete the   */
                      /* operation                           */
```

See Also

DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_RESET, and MSG_SET

Capability Constants (in Chapter 8)

Capability Container: TW_ONEVALUE (in Chapter 8).

Listing of all capabilities (in Chapter 9)

DG_CONTROL / DAT_CAPABILITY / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CAPABILITY, MSG_RESET,
pCapability);
```

pCapability = A pointer to a TW_CAPABILITY structure.

Valid States

4 only

Description

Change the Current Value of the specified capability back to its power-on value and return the new Current Value.

The power-on value is the Current Value the Source started with when it entered State 4 after a DG_CONTROL / DAT_IDENTITY / MSG_OPENDS. These values are listed as TWAIN defaults (in Chapter 9). If "no default" is specified, the Source uses it preferred default value (returned from MSG_GETDEFAULT).

Application

Set the pCapability fields as follows:

```
pCapability->Cap = the CAP_xxxx or ACAP_xxxx or ICAP_xxxx identifier
pCapability->ConType = TWON_DONTCARE16
pCapability->hContainer = NULL
```

The Source will allocate the memory for the necessary container structure but the application must free it when the operation is complete and the application no longer needs to maintain the information.

Use MSG_RESET:

- To set the Current Value of the specified capability to the Source's mandatory or preferred value, and to remove any constants from the allowed values supported by the Source.

This operation may fail for a low memory condition. Either recover from a TWCC_LOWMEMORY failure by freeing memory for the Source to use so it can continue, or terminating the acquisition and notifying the user of the low memory problem.

Source

If the application requests this operation on a capability your Source does not recognize (and you're sure you've implemented all the capabilities that you're required to), disregard the operation, but return TWRC_FAILURE with TWCC_BADCAP.

If you support the capability, reset the Current Value of the capability back to its power-on value. This value must also match the TWAIN default listed in Chapter 9.

Also return the new Current Value (just like in a MSG_GETCURRENT). Fill in the fields listed below and allocate the container structure and place its handle into pCapability->hContainer. The container should be referenced by a "handle" of type TW_HANDLE.

Fill the fields in pCapability as follows:

```
pCapability->ConType = TWON_ARRAY or TWON_ONEVALUE
pCapability->hContainer = TW_HANDLE referencing a container of ConType
```

Set ConType to the container type that matches the type for this capability. Fill the fields in the container structure with the Current Value of the capability (after resetting it as stated above).

This is a memory allocation operation. It is possible for this operation to fail due to a low memory condition. Be sure to verify that the allocation is successful. If it is not, attempt to reduce the amount of memory occupied by the Source. If the allocation cannot be made return TWRC_FAILURE with TWCC_LOWMEMORY to the application and set the pCapability->hContainer handle to NULL.

Note that this operation is **only** valid in State 4, unless CAP_EXTENDEDCAPS was negotiated. Any attempt to invoke it in any other state should be disregarded, though the Source should return TWRC_FAILURE with TWCC_SEQERROR.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
  TWCC_BADCAP        /* unknown capability--Source does not  */
                     /* recognize this capability            */
  TWCC_BADDEST       /* No such Source in-session with       */
                     /* application                          */
  TWCC_LOWMEMORY     /* Not enough memory to complete the    */
                     /* operation                            */
  TWCC_SEQERROR      /* Operation invoked in invalid state   */
```

See Also

DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, and MSG_SET

Capability Constants (in Chapter 8)

Capability Containers: TW_ONEVALUE, TW_ENUMERATION, TW_RANGE, TW_ARRAY (in Chapter 8)

Listing of all capabilities (in Chapter 9)

DG_CONTROL / DAT_CAPABILITY / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CAPABILITY, MSG_SET, pCapability);
```

pCapability = A pointer to a TW_CAPABILITY structure.

Valid States

4 only (During State 4, applications can also negotiate with Sources for permission to set the value(s) of specific capabilities in States 5 and 6 through CAP_EXTENDEDCAPS.)

Description

Changes the Current Value(s) and Available Values of the specified capability to those specified by the application.

Current Values are set when the container is a TW_ONEVALUE or TW_ARRAY. Available and Current Values are set when the container is a TW_ENUMERATION or TW_RANGE.

Note: Sources are not required to allow restriction of their Available Values, however, this is strongly recommended.

Application

An application will use the setting of a capability's Current and Available Values differently depending on how the Source was enabled (DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS).

If TW_USERINTERFACE.ShowUI = TRUE

- In State 4, set the Current Value to be displayed to the user as the current value. This value will be used for acquiring the image unless changed by the user or an automatic process (such as ICAP_AUTOBRIGHT).
- In State 4, set the Available Values to restrict the settings displayed to the user and available for use by the Source.
- In State 6, get the Current Value which was chosen by the user or automatic process. This is the setting used in the upcoming transfer.

If TW_USERINTERFACE.ShowUI = FALSE

- In State 4, set the Current Value to the setting that will be used to acquire images (unless automatic settings are set to TRUE, for example: ICAP_AUTOBRIGHT).
- In State 6, get the Current Value which was chosen by any automatic processes. This is the setting used in the upcoming transfer.

If possible, use the same container type in a MSG_SET that the Source returned from a MSG_GET. Allocate the container structure. This is where you will place the value(s) you wish to have the Source set. Store the handle into pCapability->hContainer. The container must be referenced by a “handle” of type TW_HANDLE.

Set the following:

```
pCapability->ConType = TWON_ARRAY,
                       TWON_ONEVALUE,
                       TWON_ENUMERATION, or
                       TWON_RANGE
pCapability->Cap       =  CAP_xxxx designator of
                          capability of interest
pCapability->hContainer = TW_HANDLE referencing a
                          container of ConType
```

Place the value(s) that you wish the Source to use in the container. If successful, these values will supersede any previous negotiations for this capability.

The application must free the container it allocated when the operation is complete and the application no longer needs to maintain the information.

Source

Return TWRC_FAILURE / TWCC_BADCAP:

- If the application requests this operation on a capability your Source does not recognize (and you’re sure you’ve implemented all the capabilities that you’re required to). Disregard the operation.

Return TWRC_FAILURE / TWCC_BADVALUE:

- If the application requests that a value be set that lies outside the supported range of values for the capability (smaller than your minimum value or larger than your maximum value). Set the value to that which most closely approximates the requested value.
- If the application sends a container that you do not support, or do not support in a MSG_SET.
- If the application attempts to set the Available Values and the Source does not support restriction of this capability’s Available Values.

Return TWRC_CHECKSTATUS:

- If the application requests one or more values that lie within the supported range of values (but that value does not exactly match one of the supported values), set the value to the nearest supported value. The application should then do a MSG_GET to check these values.

Return TWRC_FAILURE / TWCC_SEQERROR:

- If the application sends the MSG_SET outside of State 4 and the capability has not been negotiated in CAP_EXTENDEDCAPS.

If the request is acceptable, use the container structure referenced by pCapability->hContainer to set the Current and Available Values for the capability. If the container type is TWON_ONEVALUE or TWON_ARRAY, set the Current Value for the capability to that value. If the container type is TWON_RANGE or TWON_ENUMERATION, the Source will **optionally** limit the Available Values for the capability to match those provided by the application, masking all other internal values so that the user cannot select them. Though this behavior is not mandatory, it is strongly encouraged.

Important Note: Sources should accommodate requests to limit Available Values. In the interest of adoptability for the breadth of Source manufacturers, such accommodation is not required. It is recommended, however, that the Sources do so, and that the Source's user interface be modified (from its power-on state, and when the user interface is raised) to reflect any limitation of choices implied by the newly negotiated settings.

For example, if an application can only accept black and white image data, it tells the Source of this limitation by doing a MSG_SET on ICAP_PIXELTYPE with a TW_ENUMERATION or TW_RANGE container containing only TWPT_BW (black and white).

If the Source disregards this negotiated value and fails to modify its user interface, the user may select to acquire a color image. Either the user's selection would fail (for reasons unclear to the user) or the transfer would fail (also for unclear reasons for the user). The Source should strive to prevent such situations.

Return Codes

```
TWRC_SUCCESS
TWRC_CHECKSTATUS      /* capability value(s) could not be */
                      /* matched exactly                  */
TWRC_FAILURE
   TWCC_BADCAP        /* unknown capability--Source does  */
                      /* not recognize this capability    */
   TWCC_BADDEST       /* No such Source in-session with   */
                      /* application                      */
   TWCC_BADVALUE      /* illegal value(s)--outside        */
                      /* Source's range for capability    */
   TWCC_SEQERROR      /* Operation invoked in invalid     */
                      /* state                            */
```

See Also

DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, and MSG_RESET

Capability Constants (in Chapter 8)

Capability Containers: TW_ONEVALUE, TW_ENUMERATION, TW_RANGE, TW_ARRAY (in Chapter 8)

Listing of all capabilities (in Chapter 9)

DG_CONTROL / DAT_CUSTOMDSDATA / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CUSTOMDSDATA,
MSG_GET, pCustomData);
```

pCustomData = A pointer to a TW_CUSTOMDSDATA structure.

Valid States

4 only

Description

This operation is used by the application to query the data source for its current settings, e.g. DPI, paper size, color format. The sources settings will be returned in a TW_CUSTOMDSDATA structure. The actual format of the data in this structure is data source dependent and not defined by TWAIN.

Application

pDest references the sources identity structure. pCustomData points to a TW_CUSTOMDSDATA structure.

Source

Fills the pCustomData pointer with source specific settings. If supported, CAP_ENABLEDSUIONLY and CAP_CUSTOMDSDATA <u>are required</u>.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
    TWCC_SEQERROR
```

See Also

Capability CAP_CUSTOMDSDATA

DG_CONTROL / DAT_CUSTOMDSDATA / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CUSTOMDSDATA,
MSG_SET, pCustomData);
```

pCustomData = A pointer to a TW_CUSTOMDSDATA structure.

Valid States

4 only

Description

This operation is used by the application to set the current settings for a data source to a previous state as defined by the data contained in the pCustomData data structure. The actual format of the data in this structure is data source dependent and not defined by TWAIN.

Application

pDest references the sources identity structure. pCustomData points to a TW_CUSTOMDSDATA structure.

Source

Changes its current settings to the values specified in the pCustomData structure.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
    TWCC_SEQERROR
```

See Also

Capability CAP_CUSTOMDSDATA

DG_CONTROL / DAT_DEVICEEVENT / MSG_GET

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_DEVICEEVENT, MSG_GET,
   pSourceDeviceEvent);
```

pSourceDeviceEvent = A pointer to a TW_DEVICEEVENT structure

Valid States

4 through 7

Description

Upon receiving a DG_CONTROL / DAT_NULL / MSG_DEVICEEVENT from the Source, the Application must immediately make this call to obtain the event information.

Sources must queue the data for each event so that it is available for this call.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL - capability not supported.
   TWCC_SEQERROR - no events in the queue, or not in States 4 through 7.
```

See Also

DG_CONTROL / DAT_NULL / MSG_DEVICEEVENT
CAP_DEVICEEVENT

DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_EVENT, MSG_PROCESSEVENT, pEvent);
```

pEvent = A pointer to a TW_EVENT structure.

Valid States

5 through 7

Description

This operation supports the distribution of events from the application to Sources so that the Source can maintain its user interface and return messages to the application. Once the application has enabled the Source, it **must immediately** begin sending to the Source all events that enter the application's main event loop. This allows the Source to update its user interface in real-time and to return messages to the application which cause state transitions. Even if the application overrides the Source's user interface, it must forward all events once the Source has been enabled. The Source will tell the application whether or not each event belongs to the Source.

Note: Events only need to be forwarded to the Source while it is enabled.

The Source should be structured such that identification of the event's "owner" is handled before doing anything else. Further, the Source should return **immediately** if the Source isn't the owner. This convention should minimize performance concerns for the application (remember, these events are only sent while a Source is enabled—that is, just before and just after the transfer is taking place).

Application

Make pEvent->pEvent point to the EventRecord (on Macintosh) or message structure (on Windows).

Note: On return, the application should check the Return Code from DSM_Entry() for TWRC_DSEVENT or TWRC_NOTDSEVENT. If TWRC_DSEVENT is returned, the application should not process the event—it was consumed by the Source. If TWRC_NOTDSEVENT is returned, the application should process the event as it normally would.

With either of these Return Codes, the application should also check the pEvent->TWMessage and switch on the result. This is the mechanism used by the Source to notify the application that a data transfer is ready or that it should close the Source. The Source can return one of the following messages:

```
MSG_XFERREADY   /* Source has one or more images */
                /* ready to transfer             */
MSG_CLOSEDSREQ  /* Source wants to be closed,    */
                /* usually initiated by a        */
                /* user-generated event          */
MSG_NULL        /* no message for application    */
```

Source

Process this operation **immediately** and return to the application **immediately** if the event doesn't belong to you. Be aware that the application will be sending thousands of messages to you. Consider in-line processing and global flags to speed implementation.

Return Codes

```
TWRC_DSEVENT         /* Source consumed event--application*/
                     /* should not process it     */
TWRC_NOTDSEVENT      /* Event belongs to application -   */
                     /* process as usual              */
TWRC_FAILURE
    TWCC_BADDEST     /* No such Source in-session      */
                     /* with application               */
    TWCC_SEQERROR    /* Operation invoked in invalid   */
                     /* state                          */
```

See Also

DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ
DG_CONTROL / DAT_NULL / MSG_XFERREADY

Event loop information (in Chapter 3)

DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_FILESYSTEM,
   MSG_CHANGEDIRECTORY, pSourceFileSystem);
```

pSourceFileSystem = A pointer to a TW_FILESYSTEM structure

Valid States

4 only

Description

This operation selects the current device within the Source (camera, storage, etc). If the device is a TWFT_DOMAIN, then this command enters a directory that can contain TWFT_HOST files. If the device is a TWFT_HOST, then this command enters a directory that can contain TWFT_DIRECTORY files. If the device is a TWFT_DIRECTORY, then this command enters a directory that can contain TWFT_DIRECTORY or TWFT_IMAGE files.

Sources can support part or all of the storage hierarchy that is one of the following:

/Domain/Host/Directory/

/Host/Directory/…

/Directory/…

(Storage not supported)

It is permitted to mix domain, host, and directory names in the root file system of the Source. To help resolve any potential name conflict, Applications should set TW_FILESYSTEM-> FileType to the appropriate value for the topmost file. If this is not done and there is a name conflict, the Source's default behavior must be to use the file of type TWFT_DIRECTORY or TWFT_HOST, in that order.

For example, consider two files named "abc" in the root of a Source:

/abc/123 (abc is a domain)

/abc/789 (abc is a directory)

Change directory to the first one by setting FileType to TWFT_DOMAIN, or to the second one by setting FileType to TWFT_DIRECTORY. The FileType for each will be discovered while browsing the directory using DAT_GETFILEFIRST and DAT_GETFILENEXT. If the FileType is not specified, then the Source must change to the "/abc/789" directory.

Example:

A Source supports two devices: "/Camera" and "/Disk". If an application changes directory to /Camera, then it can negotiate imaging parameters and transfer images in a traditional fashion. If an application changes directory to "/Disk/abc/xyz", then it cannot negotiate imaging parameters (the images have already been captured); all it can do is browse the directory tree and transfer the images it finds.

The Application sets the new current working directory by placing in the InputName field an absolute or relative path. The Source returns the absolute path and name of the new directory in the OutputName field. The special filename dot "." can be used to retrieve the name of the current directory. The special filename dot-dot ".." can be used to change to the parent directory. Refer to the section on File Systems for more information.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL - capability not supported.
   TWCC_DENIED - operation denied (device not ready).
   TWCC_FILENOTFOUND - specified InputName does not exist.
   TWCC_SEQERROR - not state 4.
```

See Also

DG_CONTROL / DAT_FILESYSTEM / MSG_COPY
DG_CONTROL / DAT_FILESYSTEM / MSG_CREATEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_DELETE
DG_CONTROL / DAT_FILESYSTEM / MSG_FORMATMEDIA
DG_CONTROL / DAT_FILESYSTEM / MSG_GETCLOSE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETFIRSTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETINFO
DG_CONTROL / DAT_FILESYSTEM / MSG_GETNEXTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_RENAME

DG_CONTROL / DAT_FILESYSTEM / MSG_COPY

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_FILESYSTEM,
   MSG_COPY, pSourceFileSystem);
```

pSourceFileSystem = A pointer to a TW_FILESYSTEM structure

Valid States

4 only

Description

This operation copies a file or directory. Absolute and relative pathnames are supported. A file may not be overwritten with this command. If an Application wishes to do this, it must first delete the unwanted file and then reissue the Copy command.

The Application specifies the path and name of the entry to be copied in InputName. The Application specifies the new patch and name in OutputName.

It is not permitted to copy files into the root directory.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL - capability not supported.
   TWCC_DENIED - file cannot be deleted (root file, or protected
                 by Source).
   TWCC_FILEEXISTS - specified OutputName already exists.
   TWCC_FILENOTFOUND - InputName not found or OutputName invalid.
   TWCC_SEQERROR - not state 4.
```

See Also

DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_CREATEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_DELETE
DG_CONTROL / DAT_FILESYSTEM / MSG_FORMATMEDIA
DG_CONTROL / DAT_FILESYSTEM / MSG_GETCLOSE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETFIRSTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETINFO
DG_CONTROL / DAT_FILESYSTEM / MSG_GETNEXTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_RENAME

DG_CONTROL / DAT_FILESYSTEM / MSG_CREATEDIRECTORY

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_FILESYSTEM,
   MSG_CREATEDIRECTORY, pSourceFileSystem);
```

pSourceFileSystem = A pointer to a TW_FILESYSTEM structure

Valid States

4 only

Description

This operation creates a new directory within the current directory. Pathnames are not allowed, only the name of the new directory can be specified.

Example:

"abc" is valid.
"/Disk/abc" is not valid.

The Application specifies the name of the new directory in InputName.

On success, the Source returns the absolute path and name of the new directory in OutputName.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL - capability not supported.
   TWCC_DENIED - cannot create directory in current directory,
                 directories may not be created in root, or the
                 Source may opt to prevent the creation of new
                 directories in some instances, for instance if
                 the new directory would be too deep in the tree.
   TWCC_FILEEXISTS - the specified InputName already exists.
   TWCC_SEQERROR - not state 4.
```

See Also

DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_COPY
DG_CONTROL / DAT_FILESYSTEM / MSG_DELETE
DG_CONTROL / DAT_FILESYSTEM / MSG_FORMATMEDIA
DG_CONTROL / DAT_FILESYSTEM / MSG_GETCLOSE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETFIRSTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETINFO
DG_CONTROL / DAT_FILESYSTEM / MSG_GETNEXTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_RENAME

DG_CONTROL / DAT_FILESYSTEM / MSG_DELETE

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_FILESYSTEM,
   MSG_DELETE, pSourceFileSystem);
```

pSourceFileSystem = A pointer to a TW_FILESYSTEM structure

Valid States

4 only

Description

This operation deletes a file or directory on the device. Pathnames are not allowed, only the name of the file or directory to be deleted can be specified. Recursive deletion can be specified by setting the pSourceFileSystem->Recursive to TRUE.

Example:

"abc" is valid.
"/Disk/abc" is not valid.

The Application specifies the name of the entry to be deleted in InputName. There is no return in OutputName on success.

The Application cannot delete entries in the root directory. The Application cannot delete directories unless they are empty.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL - capability not supported.
   TWCC_DENIED - file cannot be deleted (root file, or protected
                 by Source).
   TWCC_FILENOTFOUND - filename not found.
   TWCC_NOTEMPTY - directory is not empty, and cannot be deleted.
   TWCC_SEQERROR - not state 4.
```

See Also

DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_CREATEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_FORMATMEDIA
DG_CONTROL / DAT_FILESYSTEM / MSG_GETCLOSE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETFIRSTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETINFO
DG_CONTROL / DAT_FILESYSTEM / MSG_GETNEXTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_RENAME

DG_CONTROL / DAT_FILESYSTEM / MSG_FORMATMEDIA

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_FILESYSTEM,
   MSG_FORMATMEDIA, pSourceFileSystem);
```

pSourceFileSystem = A pointer to a TW_FILESYSTEM structure

Valid States

4 only

Description

This operation formats the specified storage. This operation destroys all images and sub-directories under the selected device. Use with caution.

The Application specifies the name of the device to be deleted in InputName. There is no data returned by this call.

The Application cannot format the root directory. Sources may opt to protect their media from this command, so Applications must check return and condition codes.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL - capability not supported.
   TWCC_DENIED - format denied (root directory, or protected by Source).
   TWCC_FILENOTFOUND - filename not found.
   TWCC_SEQERROR - not state 4.
```

See Also

DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_CREATEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_DELETE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETCLOSE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETFIRSTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETINFO
DG_CONTROL / DAT_FILESYSTEM / MSG_GETNEXTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_RENAME

DG_CONTROL / DAT_FILESYSTEM / MSG_GETCLOSE

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_FILESYSTEM,
   MSG_GETCLOSE, pSourceFileSystem);
```

pSourceFileSystem = A pointer to a TW_FILESYSTEM structure

Valid States

4 through 6

Description

The operation frees the Context field in pSourceFileSystem.

Every call to DG_CONTROL / DAT_FILESYSTEM / MSG_GETFIRSTFILE must be matched by a call to MSG_GETCLOSE to release the Context field in the pSourceFileSystem structure.

An Application may (erroneously) issue this operation at any time (even if a MSG_GETFIRSTFILE has not been issued yet). Sources must protect themselves from such uses. See the section on File Systems for more information on why and how this must be done.

Return Codes

```
TWRC_SUCCESS
   TWRC_FAILURE
   TWCC_BADPROTOCOL - capability not supported.
   TWCC_SEQERROR - not state 4, 5 or 6.
```

See Also

DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_CREATEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_DELETE
DG_CONTROL / DAT_FILESYSTEM / MSG_FORMATMEDIA
DG_CONTROL / DAT_FILESYSTEM / MSG_GETFIRSTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETINFO
DG_CONTROL / DAT_FILESYSTEM / MSG_GETNEXTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_RENAME

DG_CONTROL / DAT_FILESYSTEM / MSG_GETFIRSTFILE

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_FILESYSTEM,
   MSG_GETFIRSTFILE, pSourceFileSystem);
```

pSourceFileSystem = A pointer to a TW_FILESYSTEM structure

Valid States

4 through 6

Description

This operation gets the first filename in a directory, and returns information about that file (the same information that can be retrieved with MSG_GETINFO).

The Source positions the Context to point to the first filename. InputName is ignored. OutputName contains the absolute path and name of the file. If the Application enables the Source at this time, and the PendingXfers.Count is non-zero, the Application will immediately receive a MSG_XFERREADY, and the current image will be transferred.

Applications must not assume any ordering of the files delivered by the Source, with one exception: if MSG_GETFIRSTFILE is issued in the root directory, then the operation must return a TWFT_CAMERA device.

NB: "." and ".." are NEVER reported by this command.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL - capability not supported.
   TWCC_DENIED - file exists, but information about it has not
                 been returned.
   TWCC_FILENOTFOUND - directory is empty.
   TWCC_SEQERROR - not state 4, 5 or 6.
```

See Also

DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_CREATEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_DELETE
DG_CONTROL / DAT_FILESYSTEM / MSG_FORMATMEDIA
DG_CONTROL / DAT_FILESYSTEM / MSG_GETCLOSE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETINFO
DG_CONTROL / DAT_FILESYSTEM / MSG_GETNEXTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_RENAME

DG_CONTROL / DAT_FILESYSTEM / MSG_GETINFO

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_FILESYSTEM,
   MSG_GETINFO, pSourceFileSystem);
```

pSourceFileSystem = A pointer to a TW_FILESYSTEM structure

Valid States

4 through 7

Description

This operation fills the information fields in pSourceFileSystem.

InputName contains the absolute or relative path and filename of the requested file. OutputName returns the absolute path to the file.

Example *InputName*:

"abc" is valid.
"/Disk/abc" is valid.
The empty string "" returns information about the current file (if any).
"." returns information about the current directory.
".." returns information about the parent directory.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL - capability not supported.
   TWCC_DENIED -  - file exists, but information about it has not
                    been returned.
   TWCC_FILENOTFOUND - specified file does not exist.
   TWCC_SEQERROR - not state 4 - 7, or no current file.
```

See Also

DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_CREATEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_DELETE
DG_CONTROL / DAT_FILESYSTEM / MSG_FORMATMEDIA
DG_CONTROL / DAT_FILESYSTEM / MSG_GETCLOSE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETFIRSTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_RENAME

DG_CONTROL / DAT_FILESYSTEM / MSG_GETNEXTFILE

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_FILESYSTEM,
   MSG_GETNEXTFILE, pSourceFileSystem);
```

pSourceFileSystem = A pointer to a TW_FILESYSTEM structure

Valid States

4 through 6

Description

This operation gets the next filename in a directory, and returns information about that file (the same information that can be retrieved with MSG_GETINFO).

The Source positions the Context to point to the next filename. InputName is ignored. OutputName contains the absolute path and name of the file. If the Application enables the Source at this time, and the PendingXfers.Count is non-zero, the Application will immediately receive a MSG_XFERREADY, and the current image will be transferred.

A call to MSG_GETFIRSTFILE must be issued on a given directory before the first call to MSG_GETNEXTFILE.

NB: The "." and ".." entries are NEVER reported by this command

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL - capability not supported.
   TWCC_DENIED - file exists, but information about it has not
                 been returned.
   TWCC_FILENOTFOUND - directory is empty.
   TWCC_SEQERROR - not state 4, 5 or 6, or invalid context (must issue
                   MSG_GETFILEFIRST before calling MSG_GETNEXTFILE).
```

See Also

DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_CREATEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_DELETE
DG_CONTROL / DAT_FILESYSTEM / MSG_FORMATMEDIA
DG_CONTROL / DAT_FILESYSTEM / MSG_GETCLOSE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETFIRSTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETINFO
DG_CONTROL / DAT_FILESYSTEM / MSG_RENAME

DG_CONTROL / DAT_FILESYSTEM / MSG_RENAME

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_FILESYSTEM,
   MSG_RENAME, pSourceFileSystem);
```

pSourceFileSystem = A pointer to a TW_FILESYSTEM structure

Valid States

4 only

Description

This operation renames (and optionally moves) a file or directory. Absolute and relative path names are supported. A file may not be overwritten with this command. If an Application wishes to do this it must first delete the unwanted file, then issue the rename command.

The Application specifies the path and name of the entry to be renamed in InputName. The Application specifies the new path and name in OutputName.

Filenames in the root directory cannot be moved or renamed.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL - capability not supported.
   TWCC_DENIED - file cannot be deleted (root file, or protected
                 by Source).
   TWCC_FILEEXISTS - specified OutputName already exists.
   TWCC_FILENOTFOUND - InputName not found or OutputName invalid.
   TWCC_SEQERROR - not state 4.
```

See Also

DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_CREATEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_DELETE
DG_CONTROL / DAT_FILESYSTEM / MSG_FORMATMEDIA
DG_CONTROL / DAT_FILESYSTEM / MSG_GETCLOSE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETFIRSTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETINFO
DG_CONTROL / DAT_FILESYSTEM / MSG_GETNEXTFILE

DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS *(from Application to Source Manager)*

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_IDENTITY, MSG_CLOSEDS,
pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure.

Valid States

4 only (Transitions to State 3, if successful)

Description

When an application is finished with a Source, it must formally close the session between them using this operation. This is necessary in case the Source only supports connection with a single application (many desktop scanners will behave this way). A Source such as this cannot be accessed by other applications until its current session is terminated.

Application

Reference pSourceIdentity to the application’s copy of the TW_IDENTITY structure for the Source whose session is to be ended. The application needs to unload the Source from memory after it is closed. The process for unloading the Source is similar to that used to unload the Source Manager.

Source Manager

On Macintosh only—Closes the Source and removes it from memory, following receipt of TWRC_SUCCESS from the Source.

On Windows only—Checks its internal counter to see whether any other applications are accessing the specified Source. If so, the Source Manager takes no other action. If the closing application is the last to be accessing this Source, the Source Manager closes the Source (forwards this triplet to it) and removes it from memory, following receipt of TWRC_SUCCESS from the Source.

Upon receiving the request from the Source Manager, the Source <u>immediately</u> prepares to terminate execution.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_SEQERROR     /* Operation invoked in invalid state */
```

See Also

DG_CONTROL / DAT_IDENTITY / MSG_OPENDS

DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS

(from Source Manager to Source)

Call

```
DS_Entry(pOrigin, DG_CONTROL, DAT_IDENTITY, MSG_CLOSEDS, pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure.

Valid States

4 only (Transitions Source back to the "loaded but not open" State - approximately State 3.5)

Description

Closes the Source so it can be unloaded from memory. The Source responds by doing its shutdown and clean-up activities needed to ensure the heap will be "clean" after the Source is unloaded. Under Windows, the Source will only be unloaded if the connection with the last application accessing it is about to be broken. The Source will know this by its internal "connect count" that should be maintained by any Source that supports multiple application connects.

Source Manager

pSourceIdentity is filled from a previous MSG_OPENDS operation.

Source

Perform all necessary housekeeping in anticipation of being unloaded. Be sure to dispose of any memory buffers that the Source has allocated locally, or that may have become the Source's responsibility during the course of the TWAIN session. The Source exists in a shared memory environment. It is therefore critical that all remnants of the Source, except the entry point (initial) code, be removed as the Source prepares to be unloaded.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_OPERATIONERROR   /* Internal Source error; */
                         /* handled by the Source  */
```

See Also

DG_CONTROL / DAT_IDENTITY / MSG_OPENDS

DG_CONTROL / DAT_IDENTITY / MSG_GET

(from Source Manager to Source)

Call

```
DS_Entry(pOrigin, DG_CONTROL, DAT_IDENTITY, MSG_GET, pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure.

Valid States

3 through 7 (Yes, the Source must be able to return the identity before it is opened.)

Description

This operation triplet is generated only by the Source Manager and is sent to the Source. It returns the identity structure for the Source.

Source Manager

No special set up or action required.

Source

Fills in all fields of pSourceIdentity except the Id field which is only modified by the Source Manager. This structure was allocated by either the application or the Source Manager depending on which one initiated the MSG_OPENDS operation for the Source.

Note: Sources should locate the code that handles initialization of the Source (responding to MSG_OPENDS) and identification (DAT_IDENTITY / MSG_GET) in the segment first loaded when the DLL/code resource is invoked. Responding to the identification operation should not cause any other segments to be loaded. Code to handle all other operations and to support the user interface should be located in code segments that will be loaded upon demand. Remember, the Source is a “guest” of the application and needs to be sensitive to use of available memory and other system resources. The Source Manager’s perceived performance may be adversely affected unless the Source efficiently handles identification requests.

Return Codes

```
TWRC_SUCCESS       /* This operation must succeed. */
```

DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_IDENTITY, MSG_GETDEFAULT,
pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure.

Valid States

3 through 7

Description

Gets the identification information of the system default Source.

Application

No special set up or action required.

Source Manager

Fills the structure pointed to by pSourceIdentity with identifying information about the system default Source.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_NODS          /* no Sources found matching       */
                      /* application's SupportedGroups   */
   TWCC_LOWMEMORY     /* not enough memory to perform    */
                      /* this operation                  */
```

See Also

DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST
DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT
DG_CONTROL / DAT_IDENTITY / MSG_OPENDS
DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_IDENTITY, MSG_GETFIRST,
pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure.

Valid States

3 through 7

Description

The application may obtain a list of all Sources that are currently available on the system which match the application's supported groups (DGs, that the application specified in the SupportedGroups field of its TW_IDENTITY structure). To obtain the complete list of all available Sources requires invocation of a series of operations. The first operation uses MSG_GETFIRST to find the first Source on "the list" (whichever Source the Source Manager finds first). All the following operations use DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT to get the identity information, one at a time, of all remaining Sources.

Note: The application must invoke the MSG_GETFIRST operation before a MSG_GETNEXT operation. If the MSG_GETNEXT is invoked first, the Source Manager will fail the operation (TWRC_ENDOFLIST).

If the application wants to cause a specific Source to be opened, one whose ProductName the application knows, it must first establish the existence of the Source using the MSG_GETFIRST / MSG_GETNEXT operations. Once the application has verified that the Source is available, it can request that the Source Manager open the Source using DG_CONTROL / DAT_IDENTITY / MSG_OPENDS. The application must not execute this operation without first verifying the existence of the Source because the results may be unpredictable.

Application

No special set up or action required.

Source Manager

Fills the TW_IDENTITY structure pointed to by pSourceIdentity with the identity information of the first Source found by the Source Manager within the TWAIN directory / folder.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_NODS           /* No Sources can be found      */
   TWCC_LOWMEMORY      /* Not enough memory to perform */
                       /* this operation               */
```

See Also

DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT
DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT
DG_CONTROL / DAT_IDENTITY / MSG_OPENDS
DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_IDENTITY, MSG_GETNEXT,
pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure.

Valid States

3 through 7

Description

The application may obtain a list of all Sources that are currently available on the system which match the application's supported groups (DGs, that the application specified in the SupportedGroups field of its TW_IDENTITY structure). To obtain the complete list of all available Sources requires invocation of a series of operations. The first operation uses DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST to find the first Source on "the list" (whichever Source the Source Manager finds first). All the following operations use MSG_GETNEXT to get the identity information, one at a time, of all remaining Sources.

Note: The application must invoke the MSG_GETFIRST operation before a MSG_GETNEXT operation. If the MSG_GETNEXT is invoked first, the Source Manager will fail the operation (TWRC_ENDOFLIST).

If the application wants to cause a specific Source to be opened, one whose ProductName the application knows, it must first establish the existence of the Source using the MSG_GETFIRST / MSG_GETNEXT operations. Once the application has verified that the Source is available, it can request that the Source Manager open the Source using DG_CONTROL / DAT_IDENTITY / MSG_OPENDS. The application must not execute this operation without first verifying the existence of the Source because the results may be unpredictable.

Application

No special set up or action required.

Source Manager

Fills the TW_IDENTITY structure pointed to by pSourceIdentity with the identity information of the next Source found by the Source Manager within the TWAIN directory / folder.

Return Codes

```
TWRC_SUCCESS
TWRC_ENDOFLIST        /* after MSG_GETNEXT if no more */
                      /* Sources                      */
TWRC_FAILURE
   TWCC_LOWMEMORY     /* not enough memory to perform */
                      /* this operation               */
```

See Also

DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT
DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST
DG_CONTROL / DAT_IDENTITY / MSG_OPENDS
DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

DG_CONTROL / DAT_IDENTITY / MSG_OPENDS

(from Application to Source Manager)

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_IDENTITY, MSG_OPENDS,
pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure.

Valid States

3 only (Transitions to State 4, if successful)

Description

Loads the specified Source into main memory and causes its initialization.

Application

The application may specify any available Source's TW_IDENTITY structure in pSourceIdentity. That structure may have been obtained using a MSG_GETFIRST, MSG_GETNEXT, or MSG_USERSELECT operation. If the session with the Source Manager was closed since the identity structure being used was obtained, the application must set the Id field to 0. This will cause the Source Manager to issue the Source a new Id. The application can have the Source Manager open the default Source by setting the ProductName field to "\0" (Null string) and the Id field to zero.

Source Manager

Opens the Source specified by pSourceIdentity and creates a unique Id value for this Source (under Microsoft Windows, this assumes that the Source hadn't already been opened by another application). This value is recorded in pSourceIdentity->Id. The Source Manager passes the triplet on to the Source to have the remaining fields in pSourceIdentity filled in.

Upon receiving the request from the Source Manager, the Source fills in all the fields in pSourceIdentity except for Id. If an application tries to connect to a Source that is already connected to its maximum number of applications, the Source returns TWRC_FAILURE/TWCC_MAXCONNECTIONS.

Warning: The Source and application **must** not assume that the value written into pSourceIdentity.Id will remain constant between sessions. This value is used internally by the Source Manager to uniquely identify applications and Sources and to manage the connections between them. During a different session, this value may still be valid but might be assigned to a different application or Source! Don't use this value directly.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_LOWMEMORY        /* not enough memory to */
                         /* open the Source      */
   TWCC_MAXCONNECTIONS   /* Source cannot support*/
                         /* another connection   */
   TWCC_NODS             /* specified Source was */
                         /* not found            */
   TWCC_OPERATIONERROR   /* internal Source error;*/
                         /* handled by the Source */
```

See Also

DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS
DG_CONTROL / DAT_IDENTITY / MSG_GET
DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT
DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST
DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT
DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

DG_CONTROL / DAT_IDENTITY / MSG_OPENDS

(from Source Manager to Source)

Call

```
DS_Entry(pOrigin, DG_CONTROL, DAT_IDENTITY, MSG_OPENDS, pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure.

Valid States

Source is loaded but not yet open (approximately State 3.5, session transitions to State 4, if successful).

Description

Opens the Source for operation.

Source Manager

pSourceIdentity is filled in from a previous DG_CONTROL / DAT_IDENTITY / MSG_GET and the Id field should be filled in by the Source Manager.

Source

Initializes any needed internal structures, performs necessary checks, and loads all resources needed for normal operation.

Windows only: Source should record a copy of *pOrigin, the application's TW_IDENTITY structure, whose Id field maintains a unique number identifying the application that is calling. Sources that support only a single connection should examine pOrigin->Id for each operation to verify they are being called by the application they acknowledge being connected with. All requests from other applications should fail (TWRC_FAILURE / TWCC_MAXCONNECTIONS). The Source is responsible for managing this, not the Source Manager (the Source Manager does not know in advance how many connections the Source will support).

Macintosh Note: Since the Source(s) and the Source Manager connected to a particular application live within that application's heap space, and are not shared with any other application, the discussion about multiply-connected Sources and verifying which application is invoking an operation is not relevant. A Source or Source Manager on the Macintosh can only be connected to a single application, though multiple copies of a Source or the Source Manager may be active on the same host simultaneously. These instances simply exist in different applications' heaps. If the instances need to communicate with one another, they might use a special resource file or may use the program's resource file on disk. The Source Manager manages its various instances in this way.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_LOWMEMORY          /* not enough memory to  */
                           /* open the Source        */
   TWCC_MAXCONNECTIONS     /* Source cannot support */
                           /* another connection    */
   TWCC_OPERATIONERROR     /* internal Source error;*/
                           /* handled by the Source */
```

See Also

DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS
DG_CONTROL / DAT_IDENTITY / MSG_GET

DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_IDENTITY, MSG_USERSELECT,
pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure.

Valid States

3 through 7

Description

This operation should be invoked when the user chooses **Select Source...** from the application's **File** menu (or an equivalent user action). This operation causes the Source Manager to display the Select Source dialog. This dialog allows the user to pick which Source will be used during subsequent **Acquire** operations. The Source selected becomes the system default Source. This default persists until a different Source is selected by the user. The system default Source may be overridden by an application (the override is local to only that application). Only Sources that can supply data matching one or more of the application's SupportedGroups (from the application's identity structure) will be selectable. All others will be unavailable for selection.

Application

If the application wants a particular Source, other than the system default, to be highlighted in the Select Source dialog, it should set the ProductName field of the structure pointed to by pSourceIdentity to the ProductName of that Source. This information should have been obtained from an earlier operation using DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST, MSG_GETNEXT, or MSG_USERSELECT. Otherwise, the application should set the ProductName field in pSourceIdentity to the null string ("\0"). In either case, the application should set the Id field in pSourceIdentity to zero.

If the Source Manager can't find a Source whose ProductName matches that specified by the application, it will select the system default Source (the default that matches the SupportedGroups of the application). This is not considered to be an error condition. No error will be reported. The application should check the ProductName field of pSourceIdentity following this operation to verify that the Source it wanted was opened.

Source Manager

The Source Manager displays the Select Source dialog and allows the user to select a Source. When the user clicks the "OK" button ("Select" button in the Microsoft Windows Source Manager) in the Select Source dialog, the system default Source (maintained by the Source Manager) will be changed to the selected Source. This Source's identifying information will be written into pSourceIdentity.

The “Select” button (“OK” button in the Macintosh Source Manager) will be grayed out if there are no Sources available matching the SupportedGroups specified in the application’s identity structure, pOrigin. The user must click the “Cancel” button to exit the Select Source dialog. The application cannot discern from this Return Code whether the user simply canceled the selection or there were no Sources for the user to select. If the application really wants to know whether any Sources are available that match the specified SupportedGroups it can invoke a MSG_GETFIRST operation and check for a successful result.

It copies the TW_IDENTITY structure of the selected Source into pSourceIdentity.

Suggestion for Source Developers: The string written in the Source’s TW_IDENTITY.ProductName field should clearly and unambiguously identify your product or the Source to the user (if the Source can be used to control more than one device). ProductName contains the string that will be placed in the Select Source dialog (accompanied, on the Macintosh, with an icon from the Source’s resource file representing the Source). It is further suggested that the Source’s disk file name approximate the ProductName to assist the user in equating the two.

Return Codes

```
TWRC_SUCCESS
TWRC_CANCEL         /* User clicked cancel button - maybe there   */
                    /* were no Sources                            */
TWRC_FAILURE
   TWCC_LOWMEMORY   /* not enough memory to perform this  */
                    /* operation                          */
```

See Also

DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT
DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST
DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT
DG_CONTROL / DAT_IDENTITY / MSG_OPENDS

DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ

(from Source to Application - Windows only)

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_NULL, MSG_CLOSEDSREQ, NULL);
```

This operation requires no data (NULL).

Valid States

5 through 7 (This operation causes the session to transition to State 5.)

Description

While the Source is enabled, the application is sending all events/messages to the Source. The Source will use one of these events/messages to indicate to the application that it needs to be closed.

On Windows, the Source sends this DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ to the Source Manager to cause the Source Manager to post a private message to the application's event/message loop. This guarantees that the application will have an event/message to pass to the Source Manager so it will be able to communicate the Source's Close request back to the application.

On Macintosh, the application simply sends Null events to the Source periodically to ensure it has a communication carrier when needed. Therefore, this operation is not used on a Macintosh implementation.

Source (on Windows only)

Source creates this triplet with NULL data and sends it to the Source Manager via the Source Manager's DSM_Entry point.

pDest is the TW_IDENTITY structure of the application.

Source Manager (on Windows only)

Upon receiving this triplet, the Source Manager posts a private message to the application's event/message loop. Since the application is forwarding all events/messages to the Source while the Source is enabled, this creates a communication device needed by the Source. When this private message is received by the Source Manager (via the DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation), the Source Manager will insert a MSG_CLOSEDSREQ into the TWMessage field on behalf of the Source.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_SEQERROR    /* Operation invoked in invalid state */
   TWCC_BADDEST     /* No such application in session with*/
                    /* Source                             */
```

See Also

DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT
DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS

DG_CONTROL / DAT_NULL / MSG_DEVICEEVENT

(from Source to Application)

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_NULL, MSG_DEVICEEVENT, NULL);
```

This operation requires no data (NULL)

Valid States

4 through 7

Description

When enabled the source sends this message to the Application to alert it that some event has taken place. Upon receiving this message, the Application must immediately issue a call to DG_CONTROL / DAT_DEVICEEVENT / MSG_GET to obtain the event information.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_SEQERROR - operation invoked in invalid state.
   TWCC_BADDEST - no such application in session with Source.
```

See Also

DG_CONTROL / DAT_DEVICEEVENT / MSG_GET

Capability - CAP_DEVICEEVENT

DG_CONTROL / DAT_NULL / MSG_XFERREADY *(from Source to Application - applies to Windows only)*

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_NULL, MSG_XFERREADY, NULL);
```

This operation requires no data (NULL).

Valid States

5 only (This operation causes the transition to State 6.)

Description

While the Source is enabled, the application is sending all events/messages to the Source. The Source will use one of these events/ messages to indicate to the application that the data is ready to be transferred.

On Windows, the Source sends this DG_CONTROL / DAT_NULL / MSG_XFERREADY to the Source Manager to cause the Source Manager to post a private message to the application's event/message loop. This guarantees that the application will have an event/message to pass to the Source and the Source will be able to communicate its "transfer ready" announcement back to the application.

On Macintosh, the application simply sends Null events to the Source periodically to ensure it has a communication carrier when needed. Therefore, this operation is not used on a Macintosh implementation.

Source (on Windows only)

Source creates this triplet with NULL data and sends it to the Source Manager via the Source Manager's DSM_Entry point.

pDest is the TW_IDENTITY structure of the application.

Source Manager

Upon receiving this triplet, the Source Manager posts a private message to the application's event/message loop. Since the application is forwarding all events/messages to the Source while the Source is enabled, this creates a communication device needed by the Source. When this private message is received by the Source Manager (via the DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation), the Source Manager will insert the MSG_XFERREADY into the TWMessage field on behalf of the Source.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_SEQERROR     /* Operation invoked in invalid state */
   TWCC_BADDEST      /* No such application in session with*/
                     /* Source                             */
```

See Also

DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT
DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET
DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET
DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET

DG_CONTROL / DAT_PARENT / MSG_CLOSEDSM

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_PARENT, MSG_CLOSEDSM, pParent);
```

On Windows - pParent = points to the window handle (hWnd) that will act as the Source's "parent". The variable is of type TW_INT32 and the low word of this variable must contain the window handle.

On Macintosh - pParent = should be a 32-bit NULL value.

Valid States

3 **only** (causes transition back to State 2, if successful)

Description

When the application has closed all the Sources it had previously opened, and is finished with the Source Manager (the application plans to initiate no other TWAIN sessions), it must close the Source Manager. The application should unload the Source Manager DLL or code resource after the Source Manager is closed—unless the application has immediate plans to use the Source Manager again.

Application

References the same pParent parameter that was used during the "open Source Manager" operation. If the operation returns TWRC_SUCCESS, the application should unload the Source Manager from memory.

Source Manager

Does any housekeeping needed to prepare for being unloaded from memory. This housekeeping is transparent to the application.

Windows only—If the Source Manager is open to at least one other application, it will clean up just activities relative to the closing application, then return TWRC_SUCCESS. The application will attempt to unload the Source Manager DLL. Windows will tell the application that the unload was successful, but the Source Manager will remain active and connected to the other application(s).

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_SEQERROR      /* Operation invoked in invalid state */
```

See Also

DG_CONTROL / DAT_PARENT / MSG_OPENDSM

DG_CONTROL / DAT_PARENT / MSG_OPENDSM

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_PARENT, MSG_OPENDSM, pParent);
```

On Windows - pParent = points to the window handle (hWnd) that will act as the Source's "parent". The variable is of type TW_INT32 and the low word of this variable must contain the window handle.

On Macintosh - pParent = should be a 32-bit NULL value.

Valid States

2 **only** (causes transition to State 3, if successful)

Description

Causes the Source Manager to initialize itself. This operation **must** be executed before any other operations will be accepted by the Source Manager.

Application

Windows only—The application should set the pParent parameter to point to a window handle (hWnd) of an open window that will remain open until the Source Manager is closed. If application can't open the Source Manager DLL, Windows displays an error box (this error box can be disabled by a prior call to SetErrorMode (SET_NOOPENFILEERRORBOX)).

Macintosh only—Set pParent to NULL.

Source Manager

Initializes and prepares itself for subsequent operations. Maintains a copy of pParent.

Windows only—If Source Manager is already open, Source Manager won't reinitialize but will retain a copy of pParent.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_LOWMEMORY      /* not enough memory to perform */
                       /* this operation               */
   TWCC_SEQERROR       /* Operation invoked in invalid */
                       /* state                        */
```

See Also

DG_CONTROL / DAT_PARENT / MSG_CLOSEDSM

DG_CONTROL / DAT_PASSTHRU / MSG_PASSTHRU

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_PASSTHRU,MSG_PASSTHRU,
   pSourcePassthru);
```

pSourcePassthru = A pointer to a TW_PASSTHRU structure

Valid States

4 through 7

Description

PASSTHRU is intended for the use of Source writers writing diagnostic applications. It allows raw communication with the currently selected device in the Source.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL - capability not supported.
   TWCC_SEQERROR - command could not be completed in this state.
```

See Also

CAP_PASSTHRU

DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_PENDINGXFERS, MSG_ENDXFER,
   pPendingXfers);
```

pPendingXfers = A pointer to a TW_PENDINGXFERS structure

Valid States

6 and 7

When DAT_XFERGROUP is set to DG_IMAGE:

(Transitions to State 5 if this was the last transfer (pPendingXfers->Count == 0). Transitions to State 6 if there are more transfers pending (pPendingXfers->Count != 0). To abort all remaining transfers and transition from State 6 to State 5, use DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET.

When DAT_XFERGROUP is set to DG_AUDIO:

Transitions to State 6 no matter what the value of pPendingXfers->Count.

Description

This triplet is used to cancel or terminate a transfer. Issued in state 6, this triplet cancels the next pending transfer, discards the transfer data, and decrements the pending transfers count. In state 7, this triplet terminates the current transfer. If any data has not been transferred (this is only possible during a memory transfer) that data is discarded.

The application can use this operation to cancel the next pending transfer (Source writers take note of this). For example, after the application checks TW_IMAGEINFO (or TW_AUDIOINFO, if transferring audio snippets), it may decide to not transfer the next image. The operation must be sent prior to the beginning of the transfer, otherwise the Source will simply abort the current transfer. The Source decrements the number of pending transfers.

Application

The application must invoke this operation at the end of every transfer to signal the Source that the application has received all the data it expected. The application should send this after receiving a TWRC_XFERDONE or TWRC_CANCEL.

No special set up or action required. Be sure to correctly track which state the Source will be in as a result of your action. Be aware of the value in pPendingXfers->Count both before and after the operation. Invoking this operation causes the loss of data that your user may not expect to be lost. Be very careful and prudent when using this operation.

Source

Option #1) Fill pPendingXfers->Count with the number of transfers the Source is ready to supply to the application, upon demand. If pPendingXfers->Count > 0 (or equals -1), transition to State 6 and await initiation of the next transfer by the application. If pPendingXfers->Count == 0, transition all the way back to State 5 and await the next acquisition.

Option #2) Preempt the acquired data that is next in line for transfer to the application (pending transfers can be thought of as being pushed onto a FIFO queue as acquired and popped off the queue when transferred). Decrement pPendingXfers->Count. If already acquired, discard the data for the preempted transfer. Update pPendingXfers->Count with the new number of pending transfers. If this value is indeterminate, leave the value in this field at -1. Note: -1 is not a valid value for the number of audio snippets.

Option #3) Cancel the current transfer. Discard any local buffers or data involved in the transfer. Prepare the Source and the device for the next transfer. Decrement pPendingXfers->Count (donÆt decrement if already zero or -1). If there is a transfer pending, return to State 6 and prepare the Source to begin the next transfer. If no transfer is pending, return to State 5 and await initiation of the next acquisition from the application or the user. Note: when DAT_XFERGROUP is set to DG_AUDIO, the Source will not go lower than State 2 based on the value of pPendingXfers->Count.

Note: If a Source supports simultaneous connections to more than one application, the Source should maintain a separate pPendingXfers structure for each application it is in-session with.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST  /* No such Source in-session with application */
   TWCC_SEQERROR /* Operation invoked in invalid state         */
```

See Also

DG_AUDIO / DAT_AUDIOFILEXFER / MSG_GET
DG_AUDIO / DAT_AUDIONATIVEXFER / MSG_GET
DG_CONTROL / DAT_PENDINGXFERS / MSG_GET
DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET
DG_CONTROL / DAT_XFERGROUP / MSG_SET
DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET
DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET
DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET

Capability - CAP_XFERCOUNT

DG_CONTROL / DAT_PENDINGXFERS / MSG_GET

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_PENDINGXFERS,
   MSG_GET, pPendingXfers);
```

pPendingXfers = A pointer to a TW_PENDINGXFERS structure

Valid States

4 through 7

Description

Returns the number of transfers the Source is ready to supply to the application, upon demand. If DAT_XFERGROUP is set to DG_IMAGE, this is the number of images. If DAT_XFERGROUP is set to DG_AUDIO, this is the number of audio snippets for the current image. If there is no current image, this call must return TWRC_FAILURE / TWCC_SEQERROR.

Application

No special set up or action required.

Source

Fill pPendingXfers->Count with the number of transfers the Source is ready to supply to the application, upon demand. This value should reflect the number of complete data blocks that have already been acquired or are in the process of being acquired.

When DAT_XFERGROUP is set to DG_IMAGE:

If the Source is not sure how many transfers are pending, but is sure that the number is at least one, set pPendingXfers->Count to -1. A Source connected to a device with an automatic document feeder that cannot determine the number of pages in the feeder, or how many selections the user may make on each page, would respond in this way. A Source providing access to a series of images from a video camera or a data base may also respond this way.

When DAT_XFERGROUP is set to DG_AUDIO:

-1 is not a valid value for pPendingXfers->Count.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST   /* No such Source in-session with application */
   TWCC_SEQERROR  /* Operation invoked in invalid state         */
```

See Also

DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER
DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET
DG_CONTROL / DAT_XFERGROUP / MSG_SET

Capability - CAP_XFERCOUNT

DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_PENDINGXFERS,
   MSG_RESET, pPendingXfers);
```

pPendingXfers = A pointer to a TW_PENDINGXFERS structure

Valid States

When DAT_XFERGROUP is set to DG_IMAGE:

6 only (Transitions to State 5, if successful)

When DAT_XFERGROUP is set to DG_AUDIO:

6 only (State remains at 6)

Description

Sets the number of pending transfers in the Source to zero.

Application

When DAT_XFERGROUP is set to DG_IMAGE:

No special set up or action required. Be aware of the state transition caused by this operation. Invoking this operation causes the loss of data that your user may not expect to be lost. Be very careful and prudent when using this operation. The application may need to use this operation if an error occurs within the application that necessitates breaking off all TWAIN sessions. This will get the application, Source Manager, and Source back to State 5 together.

When DAT_XFERGROUP is set to DG_AUDIO:

The available audio snippets are discarded, but the Source remains in State 6.

Source

Set pPendingXfers->Count to zero. Discard any local buffers or data involved in any of the pending transfers.

When DAT_XFERGROUP is set to DG_IMAGE:

Return to State 5 and await initiation of the next acquisition from the application or the user.

When DAT_XFERGROUP is set to DG_AUDIO:

Remain in State 6.

Note: If a Source supports simultaneous sessions with more than one application, the Source should maintain a separate pPendingXfers structure for each application it is in-session with.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST   /* No such Source in-session with application */
   TWCC_SEQERROR  /* Operation invoked in invalid state         */
```

See Also

DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER
DG_CONTROL / DAT_PENDINGXFERS / MSG_GET
DG_CONTROL / DAT_XFERGROUP / MSG_SET

Capability - CAP_XFERCOUNT

DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_SETUPFILEXFER,
   MSG_GET, pSetupFile);
```

pSetupFile = A pointer to a TW_SETUPFILEXFER structure

Valid States

4 through 6

Description

Returns information about the file into which the Source has or will put the acquired DG_IMAGE or DG_AUDIO data.

Application

No special set up or action required.

Source

Set the following:

```
pSetupFile->Format = format of destination file
       (DG_IMAGE Constants: TWFF_TIFF, TWFF_PICT, TWFF_BMP, etc.)
       (DG_AUDIO Constants: TWAF_WAV, TWAF_AIFF, TWAF_AU, etc.)
pSetupFile->FileName = name of file
       (on Windows, include the complete path name)
pSetupFile->VRefNum = volume reference number
       (Macintosh only)
```

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST     /* No such Source in-session with application */
   TWCC_BADPROTOCOL /* Source does not support file transfer      */
   TWCC_SEQERROR    /* Operation invoked in invalid state         */
```

See Also

DG_CONTROL / DAT_SETUPFILEXFER / MSG_GETDEFAULT
DG_CONTROL / DAT_SETUPFILEXFER / MSG_RESET
DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET
DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET

Capabilities - ICAP_XFERMECH, ICAP_IMAGEFILEFORMAT,
ACAP_XFERMECH, ACAP_AUDIOFILEFORMAT

DG_CONTROL / DAT_SETUPFILEXFER / MSG_GETDEFAULT

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_SETUPFILEXFER,
   MSG_GETDEFAULT, pSetupFile);
```

pSetupFile = A pointer to a TW_SETUPFILEXFER structure

Valid States

4 through 6

Description

Returns information for the default DG_IMAGE or DG_AUDIO file.

Application

No special set up or action required.

Source

Set the following:

```
pSetupFile->Format = format of destination file
      (DG_IMAGE Constants: TWFF_TIFF, TWFF_PICT, TWFF_BMP, etc.)
      (DG_AUDIO Constants: TWAF_WAV, TWAF_AIFF, TWAF_AU, etc.)
pSetupFile->FileName = name of file
      (on Windows, include the complete path name)
pSetupFile->VRefNum = volume reference number
      (Macintosh only)
```

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST     /* No such Source in-session with application */
   TWCC_BADPROTOCOL /* Source does not support file transfer      */
   TWCC_SEQERROR    /* Operation invoked in invalid state         */
```

See Also

DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET
DG_CONTROL / DAT_SETUPFILEXFER / MSG_RESET
DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET
DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET

Capabilities - ICAP_XFERMECH, ICAP_IMAGEFILEFORMAT,
ACAP_XFERMECH, ACAP_AUDIOFILEFORMAT

DG_CONTROL / DAT_SETUPFILEXFER / MSG_RESET

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_SETUPFILEXFER,
   MSG_RESET, pSetupFile);
```

pSetupFile = A pointer to a TW_SETUPFILEXFER structure

Valid States

4 only

Description

Resets the current file information to the DG_IMAGE or DG_AUDIO default file information and returns that default information..

Application

No special set up or action required.

Source

Set the following:

```
pSetupFile->Format = format of destination file
       (DG_IMAGE Constants: TWFF_TIFF, TWFF_PICT, TWFF_BMP, etc.)
       (DG_AUDIO Constants: TWAF_WAV, TWAF_AIFF, TWAF_AU, etc.)
pSetupFile->FileName = name of file
       (on Windows, include the complete path name)
pSetupFile->VRefNum = volume reference number
       (Macintosh only)
```

Note: VRefNum should be set to reflect the default file only if it already exists). Otherwise, set this field to NULL.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST     /* No such Source in-session with application */
   TWCC_BADPROTOCOL /* Source does not support file transfer      */
   TWCC_SEQERROR    /* Operation invoked in invalid state         */
```

See Also

DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET
DG_CONTROL / DAT_SETUPFILEXFER / MSG_GETDEFAULT
DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET
DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET

Capabilities - ICAP_XFERMECH, ICAP_IMAGEFILEFORMAT, ACAP_XFERMECH, ACAP_AUDIOFILEFORMAT

DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET

Call

```
DSM_Entry (pOrigin, pDest, DG_CONTROL, DAT_SETUPFILEXFER,
   MSG_SET, pSetupFile);
```

pSetupFile = A pointer to a TW_SETUPFILEXFER structure

Valid States

4 through 6

Description

Sets the file transfer information for the next file transfer. The application is responsible for verifying that the specified file name is valid and that the file either does not currently exist (in which case, the Source is to create the file), or that the existing file is available for opening and read / write operations. The application should also assure that the file format it is requesting can be provided by the Source (otherwise, the Source will generate a TWRC_FAILURE / TWCC_BADVALUE error).

Application

Set the following:

```
pSetupFile->Format = format of destination file
        (DG_IMAGE Constants: TWFF_TIFF, TWFF_PICT, TWFF_BMP, etc.)
        (DG_AUDIO Constants: TWAF_WAV, TWAF_AIFF, TWAF_AU, etc.)
pSetupFile->FileName = name of file
        (on Windows, include the complete path name)
pSetupFile->VRefNum = volume reference number
        (Macintosh only)
```

Note: ICAP_XFERMECH or ACAP_XFERMECH (depending on the value of DAT_XFERGROUP) must have been set to TWSX_FILE during previous capability negotiation.

Source

Use the specified file format and file name information to transfer the next file to the application. If any part of the information being set is wrong or missing, use the SourceÆs default file (TWAIN.TMP in the current directory for DG_IMAGE data, or TWAIN.AUD in the current directory for DG_AUDIO data) and return TWRC_FAILURE with TWCC_BADVALUE. If the format and file name are OK, but a file error occurs when trying to open the file (other than "file does not existö), return TWCC_BADVALUE and set up to use the default file. If the specified file does not exit, create it. If the file exists and has data in it, overwrite the existing data starting with the first byte of the file.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST     /* No such Source in-session with application */
   TWCC_BADPROTOCOL /* Source does not support file transfer      */
   TWCC_BADVALUE    /* Source cannot comply with one of the       */
                    /* settings                                   */
   TWCC_SEQERROR    /* Operation invoked in invalid state         */
```

See Also

DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET
DG_CONTROL / DAT_SETUPFILEXFER / MSG_GETDEFAULT
DG_CONTROL / DAT_SETUPFILEXFER / MSG_RESET
DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET

Capabilities - ICAP_XFERMECH, ICAP_IMAGEFILEFORMAT,
ACAP_XFERMECH, ACAP_AUDIOFILEFORMAT

DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_SETUPMEMXFER, MSG_GET, pSetupMem);
```

pSetupMem = A pointer to a TW_SETUPMEMXFER structure.

Valid States

4 through 6

Description

Returns the Source's preferred, minimum, and maximum allocation sizes for transfer memory buffers. The application using buffered memory transfers must use a buffer size between MinBufSize and MaxBufSize in their TW_IMAGEMEMXFER.Memory.Length when using the DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET operation. Sources may return a more efficient preferred value in State 6 after the image size, etc. has been specified.

Application

No special set up or action required.

Source

Set the following:

pSetupMem->MinBufSize = minimum usable buffer size,
in bytes

pSetupMem->MaxBufSize = maximum usable buffer size,
in bytes (-1 means an indeterminately large buffer is acceptable)

pSetupMem->Preferred = preferred transfer buffer size, in bytes

If the Source doesn't care about the size of any of these specifications, set the field(s) to TWON_DONTCARE32. This signals the application that any value for that field is OK with the Source.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST     /* No such Source in-session with */
                    /* application                    */
   TWCC_SEQERROR    /* Operation invoked in invalid   */
                    /* state                          */
```

See Also

DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET

Capabilities - ICAP_COMPRESSION, ICAP_XFERMECH

DG_CONTROL / DAT_STATUS / MSG_GET

(from Application to Source Manager)

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_STATUS, MSG_GET, pSourceStatus);
```

pSourceStatus = A pointer to a TW_STATUS structure.

Valid States

2 through 7

Description

Returns the current Condition Code for the Source Manager.

Application

NULL references the operation to the Source Manager.

Source Manager

Fills pSourceStatus->ConditionCode with its current Condition Code. Then, it will clear its internal Condition Code so you cannot issue a status inquiry twice for the same error (the information is lost after the first request).

Return Codes

```
TWRC_SUCCESS       /* This operation must succeed     */
TWRC_FAILURE
   TWCC_BADDEST    /* No such Source in-session with */
                   /* application                    */
```

See Also

Return Codes and Condition Codes (Chapter 10)

DG_CONTROL / DAT_STATUS / MSG_GET

(from Application to Source)

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_STATUS, MSG_GET, pSourceStatus);
```

pSourceStatus = A pointer to a TW_STATUS structure.

Valid States

4 through 7

Description

Returns the current Condition Code for the specified Source.

Application

pDest references a copy of the targeted Source's identity structure.

Source

Fills pSourceStatus->ConditionCode with its current Condition Code. Then, it will clear its internal Condition Code so you cannot issue a status inquiry twice for the same error (the information is lost after the first request).

Return Codes

```
TWRC_SUCCESS        /* This operation must succeed     */
TWRC_FAILURE
   TWCC_BADDEST     /* No such Source in-session with */
                    /* application                    */
```

See Also

Return Codes and Condition Codes (Chapter 10)

DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_USERINTERFACE, MSG_DISABLEDS,
pUserInterface);
```

pUserInterface = A pointer to a TW_USERINTERFACE structure.

Valid States

5 only (Transitions to State 4, if successful)

Description

This operation causes the Source's user interface, if displayed during the DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS operation, to be lowered. The Source is returned to State 4, where capability negotiation can again occur. The application can invoke this operation either because it wants to shut down the current session, or in response to the Source "posting" a MSG_CLOSEDSREQ event to it. Rarely, the application may need to close the Source because an error condition was detected.

Application

References the same pUserInterface structure as during the MSG_ENABLEDS operation. This implies that the application keep a copy of this structure locally as long as the Source is enabled.

If the application did not display the Source's built-in user interface, it will most likely invoke this operation either when all transfers have been completed or aborted (TW_PENDINGXFERS.Count = 0).

Source

If the Source's user interface is displayed, it should be lowered. The Source returns to State 4 and is again available for capability negotiation.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST       /* No such Source in-session */
                      /* with application          */
   TWCC_SEQERROR      /* Operation invoked in      */
                      /* invalid state             */
```

See Also

DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ
DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

Event loop information (in Chapter 3)

DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_USERINTERFACE, MSG_ENABLEDS,
pUserInterface);
```

pUserInterface = A pointer to a TW_USERINTERFACE structure

Valid States

4 only (Transitions to State 5, if successful)

Description

This operation causes three responses in the Source:

- Places the Source into a "ready to acquire" condition. If the application raises the Source's user interface (see #2, below), the Source will wait to assert MSG_XFERREADY until the "GO" button in its user interface or on the device is clicked. If the application bypasses the Source's user interface, this operation causes the Source to become immediately "armed". That is, the Source should assert MSG_XFERREADY as soon as it has data to transfer.
- The application can choose to raise the Source's built-in user interface, or not, using this operation. The application signals the Source's user interface should be displayed by setting pUserInterface->ShowUI to TRUE. If the application does not want the Source's user interface to be displayed, or wants to replace the Source's user interface with one of its own, it sets pUserInterface->ShowUI to FALSE. If activated, the Source's user interface will remain displayed until it is closed by the user or explicitly disabled by the application (see Note).
- Terminates Source's acceptance of "set capability" requests from the application. Capabilities can only be negotiated in State 4 (unless special arrangements are made using the CAP_EXTENDEDCAPS capability). Values of capabilities can still be inquired in States 5 through 7.

Note: Once the Source is enabled, the application **must** begin sending the Source every event that enters the application's main event loop. The application must continue to send the Source events until it disables (MSG_DISABLEDS) the Source. This is true even if the application chooses not to use the Source's built-in user interface.

Application

Set pUserInterface->ShowUI to TRUE to display the Source's built-in user interface, or to FALSE to place the Source in an "armed" condition so that it is immediately prepared to acquire data for transfer. Set ShowUI to FALSE only if bypassing the Source's built-in user interface—that is, only if the application is prepared to handle all user interaction necessary to acquire data from the selected Source.

Sources are not required to be enabled without showing their User Interface (i.e. TW_USERINTERFACE.ShowUI = FALSE). If a Source does not support ShowUI = FALSE, they will continue to be enabled just as if ShowUI = TRUE, but return TWRC_CHECKSTATUS. The application can check for this Return Code and continue knowing the Source's User Interface is being displayed.

Watch the value of pUserInterface->ModalUI after the operation has completed to see if the Source's user interface is modal or modeless.

The application must maintain a local copy of pUserInterface while the Source is enabled.

Windows only—The application should place a handle (hWnd) to the window acting as the Source's parent into pUserInterface->hParent.

Macintosh only—Set pUserInterface->hParent to NULL.

Note: Application should establish that the Source can supply compatible ICAP_PIXELTYPEs and ICAP_BITDEPTHs prior to enabling the Source. The application **must** verify that the Source can supply data of a type it can consume. If this operation fails, the application should notify the user that the device and application are incompatible due to data type mismatch. If the application diligently sets SupportedGroups in its identity structure before it tries to open the Source, the Source Manager will, in the Select Source dialog or through the MSG_GETFIRST / MSG_GETNEXT mechanism, filter out the Sources that don't match these SupportedGroups.

Source

If pUserInterface->ShowUI is TRUE, the Source should display its user interface and wait for the user to initiate an acquisition. If pUserInterface->ShowUI is FALSE, the Source should immediately begin acquiring data based on its current configuration (a device that requires the user to push a button on the device, such as a hand-scanner, will be "armed" by this operation and will assert MSG_XFERREADY as soon as the Source has data ready for transfer). The Source should fail any attempt to set a capability value (TWRC_FAILURE / TWCC_SEQERROR) until it returns to State 4 (unless an exception approval exists via a CAP_EXTENDEDCAPS agreement).

Set pUserInterface->ModalUI to TRUE if your built-in user interface is modal. Otherwise, set it to FALSE.

Note: While the Source's user interface is raised, the Source is responsible for presenting the user with appropriate progress indicators regarding the acquisition and transfer processes unless the application has set CAP_INDICATORS to FALSE. The Source must also report errors to the user (without regard for the settings of CAP_INDICATORS and ShowUI, i.e. they may be set to FALSE and errors still must be reported).

It is strongly recommended that all Sources support being enabled without their User Interface if the application requests (TW_USERINTERFACE.ShowUI = FALSE). But if your Source cannot be used without its User Interface, it should enable showing the Source User Interface (just as if ShowUI = TRUE) but return TWRC_CHECKSTATUS. All Sources, however, must support the CAP_UICONTROLLABLE. This capability reports whether or not a Source allows ShowUI = FALSE. An application can use this capability to know whether the Source-supplied user interface can be suppressed before it is displayed.

Return Codes

```
TWRC_SUCCESS
TWRC_CHECKSTATUS            /* Source cannot enable     */
                            /* without User Interface   */
                            /* so it enabled with the   */
                            /* User Interface.          */
TWRC_FAILURE
   TWCC_BADDEST             /* No such Source in-session */
                            /* with application          */
   TWCC_LOWMEMORY           /* Not enough memory to open */
                            /* the Source                */
   TWCC_OPERATIONERROR      /* Internal Source error;    */
                            /* handled by the Source     */
   TWCC_SEQERROR            /* Operation invoked in      */
                            /* invalid state             */
```

See Also

DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ
DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS

Capability - CAP_INDICATORS

Event loop information (in Chapter 3)

DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDSUIONLY

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_USERINTERFACE,
MSG_ENABLEDSUIONLY, pUserInterface);
```

pUserInterface = A pointer to a TW_USERINTERFACE structure.

Valid States

4 only (transitions to State 5, if successful)

Description

This operation is very similar to DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS operation except that no image transfer will take place. This operation is used by applications that wish to display the source user interface to allow the user to manipulate the sources current settings for DPI, paper size, etc. but not acquire an image. The ShowUI member of the TW_USERINTERFACE structure is ignored since this operations only purpose is to display the source UI. The other members of the TW_USERINTERFACE structure have the same meaning as in the DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS operation.

This operation has the following effects.

1. The source transitions from state 4 to state 5. The source will display its user interface dialog but will not have a scan button (unless its only purpose is to preview the image).
2. The application must begin sending the Source every event that enters the applications main event loop. This mechanism is the same as in the MSG_ENABLEDS operation.
3. When the user hits OK or cancel from the source user interface dialog the source will transition back to state 4 and return either MSG_CLOSEDSOK or MSG_CLOSEDSREQ in the TWMessage field of the TW_EVENT structure that the application has passed along to the source.

DG_CONTROL / DAT_XFERGROUP / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_XFERGROUP, MSG_GET, pXferGroup);
```

pXferGroup = A pointer to a TW_UINT32 value.

Valid States

4 through 6

Description

Returns the Data Group (the type of data) for the upcoming transfer. The Source is required to only supply one of the DGs specified in the SupportedGroups field of pOrigin.

Application

Should have previously (during a DG_CONTROL / DAT_PARENT / MSG_OPENDSM) set pOrigin. SupportedGroups to reflect the DGs the application is interested in receiving from a Source. Since DG_xxxx identifiers are bit flags, the application can perform a bitwise OR of DG_xxxx constants of interest to build the SupportedGroups field (this is appropriate when more kinds of data than DG_IMAGE are available).

Note: Version 1.x of the Toolkit defines DG_IMAGE as the sole Data Group (DG_CONTROL is masked from any processing of SupportedGroups). Future versions of TWAIN may define support for other DGs.

Source

Set pXferGroup to the DG_xxxx constant that identifies the type of data that is ready for transfer from the Source (DG_IMAGE is the only non-custom Data Group defined in TWAIN version 1.x).

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST       /* No such Source in-session with */
                      /* application                    */
   TWCC_SEQERROR      /* Operation invoked in invalid   */
                      /* state                          */
```

DG_IMAGE / DAT_CIECOLOR / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_CIECOLOR,
          MSG_GET, pCIEColor);
```

pCIEColor = A pointer to a TW_CIECOLOR structure.

Valid States

4 through 6

Description

Background - The DAT_CIECOLOR data argument type is used to communicate the parametrics for performing a transformation from any arbitrary set of tri-stimulus values into CIE XYZ color space. Color data stored in this format is more readily manipulated mathematically than some other spaces. See Appendix A for more information about the definitions and data structures used to describe CIE color data within TWAIN.

This operation causes the Source to report the currently active parameters to be used in converting acquired color data into CIE XYZ.

Application

Prior to invoking this operation, the application should establish that the Source can provide data in CIE XYZ form. This can be determined by invoking a MSG_GET on ICAP_PIXELTYPE. If TWPT_CIEXYZ is one of the supported types, then these operations are valid. The application can specify that transfers should use the CIE XYZ space by invoking a MSG_SET operation on ICAP_PIXELTYPE using a TW_ONEVALUE container structure whose value is TWPT_CIEXYZ.

No special set up is required. Invoking this operation following the transfer (after the Source is back in State 6) will guarantee that the exact parameters used to convert the image are reported.

Source

Fill pCIEColor with the current values applied in any conversion of image data to CIE XYZ. If no values have been set by the application, fill the structure with either the values calculated for this image or the Source's default values, whichever most accurately reflect the state of the Source.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL      /* Source does not support the  */
                         /* CIE descriptors              */
   TWCC_SEQERROR         /* Operation invoked in invalid */
                         /* state                        */
```

See Also

Capability - ICAP_PIXELTYPE

Appendix A

DG_IMAGE / DAT_EXTIMAGEINFO / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_EXTIMAGEINFO,
MSG_GET,pExtImageInfo);
```

pExtImageInfo = A pointer to a TW_EXTIMAGEINFO structure.

Valid States

7 only, after receiving TWRC_XFERDONE

Description

This operation is used by the application to query the data source for extended image attributes, .e.g. bar codes found on a page. The extended image information will be returned in a TW_EXTIMAGEINFO structure.

Application

To query extended image information, set the pExtImageInfo fields as follows:

The Application will allocate memory for the necessary container structure, the source will fill the values, and then application will free it up.

```
pExtImageInfo->NumInfos = Desired number of information;
pExtImageInfo->Info[0].InfoID = TWEI_xxxx;
pExtImageInfo->Info[1].InfoID = TWEI_xxxx;
```

Source

If the application requests information that the Source does not recognize, the Source should put TWRC_INFONOTSUPPORTED in the RetCode field of TW_INFO structure.

```
pExtImageInfo->Info[0].RetCode = TWRC_INFONOTSUPPORTED;
```

If you support the capability, fill in the fields allocating extra memory if necessary. For example, for TWEI_BARCODEX:

```
pExtImageInfo->Info[0].RetCode = TWRC_SUCCESS;
pExtImageInfo->Info[0].ItemType = TWTY_UINT32;
pExtImageInfo->Info[0].NumItems = 1;
pExtImageInfo->Info[0].Item = 20;
```

For TWEI_FORMTEMPLATEMATCH:

pExtImageInfo->Info[0].RetCode = TWRC_SUCCESS;
pExtImageInfo->Info[0].ItemType = TWTY_STR255;
pExtImageInfo->Info[0].NumItems = 1;

For handle (Application set TWMF_HANDLE),

pExtImageInfo->Info[0].Item = GlobalAlloc(GHND, sizeof(TW_STR255));

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL /* Source does not support extended image          */
                    /* information                                     */
   TWCC_SEQERROR    /* Not State 7, or in State 7 but TWRC_XFERDONE    */
                    /* has not been received yet                       */
```

See Also

Capability ICAP_EXTIMAGEINFO
The section *Extended Image Information Definitions* in Chapter 8

DG_IMAGE / DAT_GRAYRESPONSE / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_GRAYRESPONSE, MSG_RESET, pResponse);
```

pResponse = A pointer to a TW_GRAYRESPONSE structure.

Valid States

4 only

Description

Background - The two DAT_GRAYRESPONSE operations allow the application to specify a transfer curve that the Source should apply to the grayscale it acquires. This curve should be applied to the data prior to transfer. The Source should maintain an "identity response curve" to be used when it is MSG_RESET.

The MSG_RESET operation causes the Source to use its "identity response curve." The identity curve causes no change in the values of the captured data when it is applied.

Application

No special action.

Source

Apply the identity response curve to all future grayscale transfers. This means that the Source will transfer the grayscale data exactly as acquired.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL      /* Source does not support      */
                         /* grayscale response curves    */
   TWCC_SEQERROR         /* Operation invoked in invalid */
                         /* state                        */
```

See Also

DG_IMAGE / DAT_GRAYRESPONSE / MSG_SET

Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_GRAYRESPONSE / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_GRAYRESPONSE, MSG_SET, pResponse);
```

pResponse = A pointer to a TW_GRAYRESPONSE structure.

Valid States

4 only

Description

Background - The two DAT_GRAYRESPONSE operations allow the application to specify a transfer curve that the Source should apply to the grayscale it acquires. This curve should be applied to the data prior to transfer. The Source should maintain an "identity response curve" to be used when it is MSG_RESET. This identity curve should cause no change in the values of the data it is applied to.

This operation causes the Source to transform any grayscale data according to the response curve specified.

Application

All three elements of the response curve for any given index should hold the same value (the curve is stored in a TW_ELEMENT8 which contains three "channels" of data). The Source may not support this operation. The application should be diligent to examine the return code from this operation.

Source

Apply the specified response curve to all future grayscale transfers. The transformation should be applied before the data is transferred.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL      /* Source does not support      */
                         /* grayscale response curves    */
   TWCC_SEQERROR         /* Operation invoked in invalid */
                         /* state                        */
```

See Also

DG_IMAGE / DAT_GRAYRESPONSE / MSG_RESET

Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGEFILEXFER, MSG_GET, NULL);
```

This operation acts on NULL data. File information can be set with the DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation.

Valid States

6 only (Transitions to State 7, if successful. Remains in State 7 until MSG_ENDXFER operation.)

Description

This operation is used to initiate the transfer of an image from the Source to the application via the disk-file transfer mechanism. It causes the transfer to begin.

Application

No special set up or action required. Application should have already invoked the DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation unless the Source's default transfer format and file name (typically, TWAIN.TMP) are acceptable to the application. The application need only invoke this operation once per image transferred.

Notes: If the application is planning to receive multiple images from the Source while using the Source's default file name, the application should plan to pause between transfers and copy the file just written. The Source will overwrite the file unless it is instructed to write to a different file.

Applications can specify a unique file for each transfer using DAT_SETUPFILEXFER / MSG_SET in State 6 or 5 (and 4, of course).

Source

Acquire the image data, format it, create any appropriate header information, and write everything into the file specified by the previous DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation, and close the file.

Handling Possible File Conditions:

- If the application did not set conditions up using the DAT_SETUPFILEXFER / MSG_SET operation during this session, use your own default file name, file format, and location for the created file.
- If the specified file already exists, overwrite the file in place.
- If the specified file does not exist, create the file.
- If the specified file exists and cannot be accessed, or a system error occurs while writing the file, report the error to the user and return TWRC_FAILURE with TWCC_OPERATIONERROR. Stay in State 6. The file contents are invalid. The image whose transfer failed is still a pending transfer so do not decrement TW_PENDINGXFERS.Count.
- If the file is written successfully, return TWRC_XFERDONE.
- If the user cancels the transfer, return TWRC_CANCEL.

Return Codes

```
TWRC_XFERDONE
TWRC_CANCEL
TWRC_FAILURE
   TWCC_BADDEST           /* No such Source in-session */
                          /* with application          */
   TWCC_OPERATIONERROR    /* Failure in the Source --  */
                          /* transfer invalid          */
   TWCC_SEQERROR          /* Operation invoked in      */
                          /* invalid state             */
```

See Also

DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET
DG_IMAGE / DAT_IMAGEINFO / MSG_GET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET,

Capabilities - ICAP_XFERMECH, ICAP_IMAGEFILEFORMAT

DG_IMAGE / DAT_IMAGEINFO / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGEINFO, MSG_GET,pImageInfo);
```

pImageInfo = A pointer to a TW_IMAGEINFO structure.

Valid States

6 and 7 (State 7 only after receiving TWRC_XFERDONE)

Description

When called in State 6, this operation provides to the application general image description information about the image about to be transferred.

When called in State 7, this operation provides the Application with specific image description information about the current image that has just been transferred. It is important during a Memory transfer to call this triplet only after TWRC_XFERDONE is received, since that is the only time the Source will know all the final image information.

The same data structure type is used regardless of the mechanism used to transfer the image (Native, Disk File, or Buffered Memory transfer).

Application

The Application can use this operation to check the parameters of the image before initiating the transfer during State 6, or to clarify image parameters during State 7 after the transfer is complete.

Applications may inform Sources that they accept -1 value for ImageHeight / ImageWidth by setting the ICAP_UNDEFINEDIMAGESIZE capability to TRUE.

Should the Application decide to invoke any Source features that allow the image description information to change during scanning (such as ICAP_UNDEFINEDIMAGESIZE) and still wish to transfer in Buffered memory mode, a DG_CONTROL / DAT_IMAGEINFO / MSG_GET call must be made in State 7 after receiving TWRC_XFERDONE to properly interpret the image data. This is not the default behavior of the Source.

Note that the speed at which the Application supplies buffers may determine the scanning speed.

Source

During State 6 - Fills in all fields in pImageInfo. All fields are filled in as you would expect with the following exceptions:

XResolution or YResolution

Set to -1 if the device creates data with no inherent resolution (such as a digital camera).

ImageWidth

Set to -1 if the image width to be acquired is unknown (such as when using a hand-held scanner and dragging left-to-right) , and the Application has set ICAP_UNDEFINEDIMAGESIZE to TRUE. In this case the Source must transfer the image in tiles.

ImageLength

ImageLength—Set to -1 if the image length to be acquired is unknown (such as when using a hand-held scanner and dragging top-to-bottom), and the Application has set ICAP_UNDEFINEDIMAGESIZE to TRUE.

During State 7 - Fills in all fields in pImageInfo. All fields are filled in as during State 6, except ImageWidth and ImageLength MUST be valid. Source shall return TWRC_SEQERROR if call is made before TWRC_XFERDONE is sent.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST     /* No such Source in-session with */
                    /* application                    */
   TWCC_SEQERROR    /* Operation invoked in invalid   */
                    /* state                          */
```

See Also

DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET
DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET
DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET

Capabilities - ICAP_BITDEPTH, ICAP_COMPRESSION, ICAP_PIXELTYPE, ICAP_PLANARCHUNKY, ICAP_XRESOLUTION, ICAP_YRESOLUTION

DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGELAYOUT, MSG_GET, pImageLayout);
```

pImageLayout = A pointer to a TW_IMAGELAYOUT structure.

Valid States

4 through 6

Description

The DAT_IMAGELAYOUT operations control information on the physical layout of the image on the acquisition platform of the Source (e.g. the glass of a flatbed scanner, the size of a photograph, etc.).

The MSG_GET operation describes both the size and placement of the image on the original "page". The coordinates on the original page and the extents of the image are expressed in the unit of measure currently negotiated for ICAP_UNITS (default is inches).

The outline of the image is expressed by a "frame." The Left, Top, Right, and Bottom edges of the frame are stored in pImageLayout->Frame. These values place the frame within the original page. All measurements are relative to the page's "upper-left" corner. Define "upper-left" by how the image would appear on the computer's screen before any rotation or other position transform is applied to the image data. This origin point will be apparent for most Sources (although folks working with satellites or radio telescopes may be at a bit of a loss).

Finally pImageLayout optionally includes information about which frame on the page, which page within a document, and which document the image belongs to. These fields were included mostly for future versions which could merge more than one type of data. A more immediate use might be for an application that needs to keep track of which frame on the page an image came from while acquiring from a Source that can supply more than one image from the same page at the same time. The information in this structure always describes the current image. To set multiple frames for any page simultaneously, reference ICAP_FRAMES.

Application

No special set up or action required, unless the current units of measure are unacceptable. In that case, the application must re-negotiate ICAP_UNITS prior to invoking this operation. Remember to do this in State 4—the only state wherein capabilities can be set or reset.

Beyond supplying possibly interesting position information on the image to be transferred, the application can use this structure to constrain the final size of the image and to relate the image within a series of pages or documents (see the DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET operation).

Source

Fill all fields of pImageLayout. Most Sources will set FrameNumber, PageNumber, and DocumentNumber to 1.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST        /* No such Source in-session    */
                       /* with application             */
   TWCC_SEQERROR       /* Operation invoked in invalid */
                       /* state                        */
```

See Also

DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT
DG_IMAGE / DAT_IMAGELAYOUT / MSG_RESET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET

Capabilities - Many such as ICAP_FRAMES, ICAP_MAXFRAMES, ICAP_UNITS

DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGELAYOUT, MSG_GETDEFAULT,
pImageLayout);
```

pImageLayout = A pointer to a TW_IMAGELAYOUT structure.

Valid States

4 through 6

Description

The DAT_IMAGELAYOUT operations control information on the physical layout of the image on the acquisition platform of the Source (e.g. the glass of a flatbed scanner, the size of a photograph, etc.).

This operation returns the default information on the layout of an image. This is the size and position of the image that will be acquired from the Source if the acquisition is started with the Source (and the device it is controlling) in its power-on state (for instance, most flatbed scanners will capture the entire bed).

Application

No special set up or action required.

Source

Fill in all fields of pImageLayout with the device's power-on origin and extents. Most Sources will set FrameNumber, PageNumber, and DocumentNumber to 1.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST        /* No such Source in-session    */
                       /* with application             */
   TWCC_SEQERROR       /* Operation invoked in invalid */
                       /* state                        */
```

See Also

DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT
DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET

Capabilities - ICAP_FRAMES, ICAP_MAXFRAMES, ICAP_UNITS

DG_IMAGE / DAT_IMAGELAYOUT / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGELAYOUT, MSG_RESET,
pImageLayout);
```

pImageLayout = A pointer to a TW_IMAGELAYOUT structure.

Valid States

4 only

Description

The DAT_IMAGELAYOUT operations control information on the physical layout of the image on the acquisition platform of the Source (e.g. the glass of a flatbed scanner, the size of a photograph, etc.).

This operation sets the image layout information for the next transfer to its default settings.

Application

No special set up or action required. Ascertain the current settings of ICAP_ORIENTATION, ICAP_PHYSICALWIDTH, and ICAP_PHYSICALHEIGHT if you don't already know this device's power-on default values.

Source

Reset all the fields of the structure pointed at by pImageLayout to the device's power-on origin and extents. There is an implied resetting of ICAP_ORIENTATION, ICAP_PHYSICALWIDTH, and ICAP_PHYSICALHEIGHT to the device's power-on default values.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST      /* No such Source in-session    */
                     /* with application             */
   TWCC_SEQERROR     /* Operation invoked in invalid */
                     /* state                        */
```

See Also

DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT
DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET

Capabilities - ICAP_FRAMES, ICAP_MAXFRAMES, ICAP_UNITS

DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGELAYOUT, MSG_SET, pImageLayout);
```

pImageLayout = A pointer to a TW_IMAGELAYOUT structure.

Valid States

4 only

Description

The DAT_IMAGELAYOUT operations control information on the physical layout of the image on the acquisition platform of the Source (e.g. the glass of a flatbed scanner, the size of a photograph, etc.).

This operation sets the layout for the next image transfer. This allows the application to specify the physical area to be acquired during the next image transfer (for instance, a frame-based application would pass to the Source the size of the frame the user selected within the application—the helpful Source would present a selection region already sized to match the layout frame size).

If the application and Source have negotiated one or more frames through ICAP_FRAMES, the frame set with this operation will only persist until the transfer following this one. Otherwise, the frame will persist as the current frame for the remainder of the session (unless superseded by negotiation on ICAP_FRAMES or another operation on DAT_IMAGELAYOUT overrides it).

The application writer should note that setting these values is a request. The Source should first try to match the requested values exactly. Failing that, it should approximate the requested values as closely as it can—extents of the approximated frame should at least equal the requested extents unless the device cannot comply. The Source should return TWRC_CHECKSTATUS if the actual values set in pImageLayout->Frame are greater than or equal to the requested values in both extents. If one or both of the requested values exceed the Source's available values, the Source should return TWRC_FAILURE with TWCC_BADVALUE. The application should check for these return codes and perform a MSG_GET to verify that the values set by the Source are acceptable. The application may choose to cancel the transfer if Source could not set the layout information closely enough to the requested values.

Application

Fill in all fields of pImageLayout. Especially important is the Frame field whose values are expressed in ICAP_UNITS. If the application doesn't care about one or more of the other fields, be sure to set them to -1 to prevent confusion. If the application only cares about the extents of the Frame, and not about the origin on the page, set the Frame.Top and Frame.Left to zero. Otherwise, the application can specify the location on the page where the Source should begin acquiring the image, in addition to the extents of the acquired image.

Source

Use the values in pImageLayout as the Source's current image layout information. If you are unable to set the device exactly to the values requested in the Frame field, set them as closely as possible, always snapping to a value that will result in a larger frame, and return TWRC_CHECKSTATUS to the application.

If the application has set Frame.Top and Frame.Left to a non-zero value , set the origin for the image to be acquired accordingly. If possible, the Source should consider reflecting these settings in the user interface when it is raised. For instance, if your Source presents a pre-scan image, consider showing the selection region in the proper location and with the proper size suggested by the settings from this operation.

If the requested values exceed the maximum size the Source can acquire, set the pImageLayout->Frame values used within the Source to the largest extent possible within the axis of the offending value. Return TWRC_FAILURE with TWCC_BADVALUE.

Return Codes

```
TWRC_SUCCESS
TWRC_CHECKSTATUS     /* Source approximated the requested*/
                     /* values                           */
TWRC_FAILURE
   TWCC_BADDEST      /* No such Source in-session       */
                     /* with application                */
   TWCC_BADVALUE     /* Specified Layout values illegal */
                     /* for Source                      */
   TWCC_SEQERROR     /* Operation invoked in invalid    */
                     /* state                           */
```

See Also

DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT
DG_IMAGE / DAT_IMAGELAYOUT / MSG_RESET

Capabilities - ICAP_FRAMES, ICAP_MAXFRAMES, ICAP_UNITS

DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGEMEMXFER, MSG_GET,
pImageMemXfer);
```

pImageMemXfer = A pointer to a TW_IMAGEMEMXFER structure.

Valid States

6 and 7 (Transitions to State 7, if successful. Remains in State 7 until MSG_ENDXFER operation.)

Description

This operation is used to initiate the transfer of an image from the Source to the application via the Buffered Memory transfer mechanism.

This operation supports the transfer of successive blocks of image data (in strips or, optionally, tiles) from the Source into one or more main memory transfer buffers. These buffers (for strips) are allocated and owned by the application. For tiled transfers, the source allocates the buffers. The application should repeatedly invoke this operation while TWRC_SUCCESS is returned by the Source.

Application

The application will allocate one or more memory buffers to contain the data being transferred from the Source. The application may allocate enough buffer space to contain the entire image being transferred or, more commonly, use the transfer buffer(s) as a temporary holding area while the complete image is assembled elsewhere (on disk, for instance).

The size of the allocated buffer(s) should be homogeneous (don’t change buffer sizes during transfer). The size the application selects should be based on the information returned by the Source from the DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET operation. The application should do its best to allocate transfer buffers of the size “preferred” by the Source. This will enhance the chances for superior transfer performance. The buffer size must be between MinBufSize and MaxBufSize as reported by the Source. Further, the buffers must contain an even number of bytes. Memory buffers must be double-word aligned and should be padded with zeros at the end of each raster line.

If the application sets up buffers that are either too small or too large, the Source will fail the operation returning TWRC_FAILURE / TWCC_BADVALUE.

Once the buffers have been set up, the application should fill pImageMemXfer->Memory.Length with the actual size (in bytes) of each memory buffer (which are, of course, all the same size).

Windows only—The buffers should be allocated in global memory.

Source

Prior to writing the first buffer, check pImageMemXfer->Memory.Length for the size of the buffer(s) the application has allocated. If the size lies outside the maximum or minimum buffer size communicated to the application during the DG_CONTROL /

DAT_SETUPMEMXFER / MSG_GET operation, return TWRC_FAILURE/TWCC_BADVALUE and remain in State 6.

If the buffer is of an acceptable size, fill in all fields of pImageMemXfer except pImageMemXfer->Memory. The Source must write the data block into the buffer referenced by pImageMemXfer->Memory.TheMem. Store the actual number of bytes written into the buffer in pImageMemXfer->BytesWritten. Compressed and tiled data effects how the Source fills in these values.

Return TWRC_SUCCESS after successfully writing each buffer. Return TWRC_CANCEL if the Source needs to terminate the transfer before the last buffer is written (as when the user aborts the transfer from the Source's user interface). Return TWRC_XFERDONE to signal that the last buffer has been written. Following completion of the transfer, either after all the data has been written or the transfer has been canceled, remain in State 7 until explicitly transitioned back to State 6 by the application (DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER).

If TWRC_FAILURE occurred on the first buffer, the session remains in State 6. If failing on a subsequent buffer, the session remains in State 7. The strip whose transfer failed is still pending.

Notes on Memory Usage: Following a canceled transfer, the Source should dispose of the image that was being transferred and assure that any temporary variable and local buffer allocations are eliminated. The Source should be wary of allocating large temporary buffers or variables. Doing so may disrupt or even disable the transfer process. The application should be aware of the possible needs of the Source to allocate such space, however, and consider allocating all large blocks of RAM needed to support the transfer prior to invoking this operation. This may be especially important for devices that create image transfers of indeterminate size—such as hand-held scanners.

Return Codes

```
TWRC_SUCCESS                /* Source done transferring       */
                            /* the specified block            */
TWRC_XFERDONE               /* Source done transferring       */
                            /* the specified image            */
TWRC_CANCEL                 /* User aborted the transfer from */
                            /* the Source                     */
TWRC_FAILURE
   TWCC_BADDEST             /* No such Source in-session */
                            /*with application           */
   TWCC_BADVALUE            /* Size of buffer did not    */
                            /* match TW_SETUPMEMXFER     */
   TWCC_OPERATIONERROR      /* Failure in the Source--   */
                            /* transfer invalid          */
   TWCC_SEQERROR            /* Operation invoked in      */
                            /* invalid state             */
```

See Also

DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET
DG_IMAGE / DAT_IMAGEINFO / MSG_GET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET

Capabilities - ICAP_COMPRESSION, ICAP_TILES, ICAP_XFERMECH

DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGENATIVEXFER, MSG_GET, pHandle);
```

pHandle = A pointer to a variable of type TW_UINT32.

Windows - This 32 bit integer is a handle variable to a DIB (Device Independent Bitmap) located in memory.

Macintosh - This 32-bit integer is a handle to a Picture (a PicHandle). It is a QuickDraw picture located in memory.

Valid States

6 only (Transitions to State 7, if successful. Remains in State 7 until MSG_ENDXFER operation).

Description

Causes the transfer of an image's data from the Source to the application, via the Native transfer mechanism, to begin. The resulting data is stored in main memory in a single block. The data is stored in Picture (PICT) format on the Macintosh and as a device-independent bitmap (DIB) under Microsoft Windows. The size of the image that can be transferred is limited to the size of the memory block that can be allocated by the Source.

Note: This is the default transfer mechanism. All Source's support this mechanism. The Source will use this mechanism unless the application explicitly negotiates a different transfer mechanism with ICAP_XFERMECH.

Application

The application need only invoke this operation once per image. The Source allocates the largest block available and transfers the image into it. If the image is too large to fit, the Source may resize the image. Read the DIB header or check the picFrame in the Picture to determine if this happened. The application is responsible for deallocating the memory block holding the Native-format image.

Windows only—Set pHandle pointing to a handle to a device-independent bit map (DIB) in memory. The Source will allocate the image buffer and return the handle to the address specified..

Macintosh only—Set pHandle pointing to a handle to a Picture in memory. The Source will allocate the image buffer at the memory location referenced by the handle.

Note: This odd combination of pointer and handle to reference the image data block was used to assure that the allocated memory object would be relocatable under Microsoft Windows, Macintosh, and UNIX. A handle was required for this task on both the Macintosh and under Microsoft Windows; though pointers are inherently relocatable under UNIX. Rather than disturb the entry points convention that the data object is always referenced by a pointer, it was decided to have that pointer reference the relocatable handle. A handle in UNIX is typecast to a pointer.

Source

Allocate a single block of memory to hold the image data and write the image data into it using the appropriate format for the operating environment. The source must assure that the allocated block will be accessible to the application. Place the handle of the allocated block in the TW_UINT32 pointed to by pHandle.

Microsoft Windows: Format the data block as a DIB. Use GlobalAlloc or equivalent under windows. Under 16 bit Microsoft Windows, place the handle in the low word of the TW_UINT32. The following assignment will work in either Win16 or Win32:

```
(HGLOBAL FAR *) pHandle = hDIB;
```

See the Windows SDK documentation under Structures: BIMAPINFO, BITMAPINFOHEADER, RGBQUAD. See also “DIBs and their use” by Ron Gery, in the Microsoft Development Library (MSDN CD).

Notes:

- Do not use BITMAPCOREINFO or BIMAPCOREHEADER as these are for OS/2 compatibility only.
- Always follow the BITMAPINFOHEADER with the color table and only save 1, 4, or 8 bit DIBs
- Color table entries are RGBQUADs, which are stored in memory as BGR not RGB.
- For 24 bit color DIBs, the “pixels” are also stored in BGR order, not RGB.
- DIBs are stored ‘upside-down’ - the first pixel in the DIB is the lower-left corner of the image, and the last pixel is the upper-right corner.
- DIBs can be larger than 64K, but be careful, a 24 bit pixel can straddle a 64K boundary!
- Pixels in 1, 4, and 8 bit DIBs are “always” color table indices, you must index through the color table to determine the color value of a pixel.

Macintosh: Format the data block as a PICT, preferably using standard system calls.

Microsoft Windows and Macintosh: If the allocation fails, it is recommended that you allow the user the option to re-size the image to fit within available memory or to cancel the transfer (assuming that the Source user interface is displayed). If the user chooses to cancel the transfer, return TWRC_CANCEL. If the user wants to re-size the image, the Source might choose to blindly crop the image, clip a selection region to the maximum supported size for the current memory configuration, or allow the user to re-acquire the image altogether. The user will usually feel more in control if you provide one or both of the last two options, but the first may make the most sense for your Source.

If the allocation fails and the image cannot be clipped, return TWRC_FAILURE and remain in State 6. Set the pHandle to NULL. The image whose transfer failed is still pending transfer. Do not decrement TW_PENDINGXFERS.Count.

Return Codes

```
TWRC_XFERDONE            /* Source done transferring the */
                         /* specified block              */
TWRC_CANCEL              /* User aborted the transfer    */
                         /* within the Source            */
TWRC_FAILURE
   TWCC_BADDEST          /* No such Source in session */
                         /* with application          */
   TWCC_LOWMEMORY        /* Not enough memory for     */
                         /* image--cannot crop to fit */
   TWCC_OPERATIONERROR   /* Failure in the Source--   */
                         /* transfer invalid          */
   TWCC_SEQERROR         /* Operation invoked in      */
                         /* invalid state             */
```

See Also

DG_IMAGE / DAT_IMAGEINFO / MSG_GET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET

Capability - ICAP_XFERMECH

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_JPEGCOMPRESSION, MSG_GET,
pCompData);
```

pCompData = A pointer to a TW_JPEGCOMPRESSION structure.

Valid States

4 through 6

Description

Causes the Source to return the parameters that will be used during the compression of data using the JPEG algorithms.

All the information that is reported by the MSG_GET operation will be available in the header portion of the JPEG data. Transferring JPEG-compressed data through memory buffers is slightly different than other types of buffered transfers. The difference is that the JPEG-compressed image data will be prefaced by a block of uncompressed information—the JPEG header. This header information contains all the information that is returned from the MSG_GET operation. The compressed image information follows the header. The Source should return the header information in the first transfer. The compressed image data will then follow in the second through the final buffer. If the application is allocating the buffers, it should assure that the buffer size for transfer of the header is large enough to contain the complete header.

Application

The application allocates the TW_JPEGCOMPRESSION structure.

Source

Fill pCompData with the parameters that will be applied to the next JPEG-compression operation. The Source must allocate memory for the contents of the pointer fields pointed to within the structure (i.e. QuantTable, HuffmanDC, and HuffmanAC).

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL       /* Source does not support JPEG */
                          /* data compression            */
   TWCC_SEQERROR          /* Operation invoked in invalid */
                          /* state                       */
```

See Also

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GETDEFAULT
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_RESET
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_SET

Capability - ICAP_COMPRESSION

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GETDEFAULT

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_JPEGCOMPRESSION, MSG_GETDEFAULT,
pCompData);
```

pCompData = A pointer to a TW_JPEGCOMPRESSION structure.

Valid States

4 through 6

Description

Causes the Source to return the power-on default values applied to JPEG-compressed data transfers.

Application

The application allocates the TW_JPEGCOMPRESSION structure.

Source

Fill in pCompData with the power-on default values. The Source must allocate memory for the contents of the pointer fields pointed to within the structure (i.e. QuantTable, HuffmanDC and HuffmanAC). The Source should maintain meaningful default values.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL        /* Source does not support JPEG */
                           /* data compression             */
   TWCC_SEQERROR           /* Operation invoked in invalid */
                           /* state                        */
```

See Also

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GET
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_RESET
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_SET

Capability - ICAP_COMPRESSION

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_JPEGCOMPRESSION, MSG_RESET,
pCompData);
```

pCompData = A pointer to a TW_JPEGCOMPRESSION structure.

Valid States

4 only

Description

Return the Source to using its power-on default values for JPEG-compressed transfers.

Application

No special action. May want to perform a MSG_GETDEFAULT if you're curious what the new values might be.

Source

Use your power-on default values for all future JPEG-compressed transfers. The Source should maintain meaningful default values for all parameters.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL       /* Source does not support JPEG */
                          /* data compression            */
   TWCC_SEQERROR          /* Operation invoked in invalid */
                          /* state                        */
```

See Also

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GET
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GETDEFAULT
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_SET

Capability - ICAP_COMPRESSION

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_JPEGCOMPRESSION, MSG_SET,
pCompData);
```

pCompData = A pointer to a TW_JPEGCOMPRESSION structure.

Valid States

4 only

Description

Allows the application to configure the compression parameters to be used on all future JPEG-compressed transfers during the current session. The application should have already established that the requested values are supported by the Source.

Application

Fill pCompData. Write TWON_DONTCARE16 into the numeric fields that don't matter to the application. Write NULL into the table fields that should use the default tables as defined by the JPEG specification.

Source

Adopt the requested values for use with all future JPEG-compressed transfers. If a value does not exactly match an available value, match the value as closely as possible and return TWRC_CHECKSTATUS. If the value is beyond the range of available values, clip to the nearest value and return TWRC_FAILURE/TWCC_BADVALUE.

Return Codes

```
TWRC_SUCCESS
TWRC_CHECKSTATUS
TWRC_FAILURE
   TWCC_BADPROTOCOL         /* Source does not support JPEG */
                            /* data compression            */
   TWCC_BADVALUE            /* illegal value specified     */
   TWCC_SEQERROR            /* Operation invoked in invalid */
                            /* state                       */
```

See Also

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GET
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GETDEFAULT
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_RESET

Capability - ICAP_COMPRESSION

DG_IMAGE / DAT_PALETTE8 / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_PALETTE8, MSG_GET, pPalette);
```

pPalette = A pointer to a TW_PALETTE8 structure.

Valid States

4 through 6

Description

This operation causes the Source to report its current palette information. The application should assure that the Source can provide palette information by invoking a MSG_GET operation on ICAP_PIXELTYPE and checking for TWPT_PALETTE. If this pixel type has not been established as the type to be used for future acquisitions, the Source should respond with its default palette.

To assure that the palette information is wholly accurate, the application should invoke this operation immediately after completion of the image transfer. The Source may perform palette optimization during acquisition of the data and the palette it reports before the transfer will differ from the one available afterwards.

(In general, the DAT_PALETTE8 operations are specialized to deal with 8-bit data, whether grayscale or color (8-bit or 24-bit). Most current devices provide data with this bit depth. These operations allow the application to inquire a Source's support for palette color data and set up a palette color transfer. See Chapter 8 for the definitions and data structures used to describe palette color data within TWAIN.)

Application

The application should allocate the pPalette structure for the Source.

Source

Fill pPalette with the current palette. If no palette has been specified or calculated, use the Source's default palette (which may coincidentally be the current or default system palette).

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL          /* Source does not support      */
                             /* palette color transfers      */
   TWCC_SEQERROR             /* Operation invoked in invalid */
                             /* state                        */
```

See Also

DG_IMAGE / DAT_PALETTE8 / MSG_GETDEFAULT
DG_IMAGE / DAT_PALETTE8 / MSG_RESET
DG_IMAGE / DAT_PALETTE8 / MSG_SET

Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_PALETTE8 / MSG_GETDEFAULT

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_PALETTE8, MSG_GETDEFAULT, pPalette);
```

pPalette = A pointer to a TW_PALETTE8 structure.

Valid States

4 through 6

Description

This operation causes the Source to report its power-on default palette.

Application

The application should allocate the pPalette structure for the Source.

Source

Fill pPalette with the default palette.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL        /* Source does not support      */
                           /* palette color transfers      */
   TWCC_SEQERROR           /* Operation invoked in invalid */
                           /* state                        */
```

See Also

DG_IMAGE / DAT_PALETTE8 / MSG_GET
DG_IMAGE / DAT_PALETTE8 / MSG_RESET
DG_IMAGE / DAT_PALETTE8 / MSG_SET

Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_PALETTE8 / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_PALETTE8, MSG_RESET, pPalette);
```

pPalette = A pointer to a TW_PALETTE8 structure.

Valid States

4 only

Description

This operation causes the Source to dispose of any current palette it has and to use its default palette for the next palette transfer. A Source that always performs palette optimization may not use the default palette for the next transfer, but should dispose of its current palette and adopt the default palette for the moment, anyway. The application can check the actual palette information by invoking a MSG_GET operation immediately following the image transfer.

Application

The application should allocate the pPalette structure for the Source.

Source

Fill pPalette with the default palette and use the default palette for the next palette transfer.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL          /* Source does not support      */
                             /* palette color transfers      */
   TWCC_SEQERROR             /* Operation invoked in invalid */
                             /* state                        */
```

See Also

DG_IMAGE / DAT_PALETTE8 / MSG_GET
DG_IMAGE / DAT_PALETTE8 / MSG_GETDEFAULT
DG_IMAGE / DAT_PALETTE8 / MSG_SET

Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_PALETTE8 / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_PALETTE8, MSG_SET, pPalette);
```

pPalette = A pointer to a TW_PALETTE8 structure.

Valid States

4 only

Description

This operation requests that the Source adopt the specified palette for use with all subsequent palette transfers. The application should be careful to supply a palette that matches the bit depth of the Source. The Source is not required to adopt this palette. The application should be careful to check the return value from this operation.

Application

Fill pPalette with the desired palette. If writing grayscale information, write the same data into the Channel1, Channel2, and Channel3 fields of the Colors array. If NumColors != 256, fill the unused array elements with minimum ("black") values.

Source

The Source should not return TWRC_SUCCESS unless it will actually use the requested palette. The Source should not modify the palette in any way until the transfer is complete. The palette should be used for all remaining palette transfers for the duration of the session.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL          /* Source does not support      */
                             /* palette color transfers      */
   TWCC_SEQERROR             /* Operation invoked in invalid */
                             /* state                        */
```

See Also

DG_IMAGE / DAT_PALETTE8 / MSG_GET
DG_IMAGE / DAT_PALETTE8 / MSG_GETDEFAULT
DG_IMAGE / DAT_PALETTE8 / MSG_RESET

Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_RGBRESPONSE / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_RGBRESPONSE, MSG_RESET, pResponse);
```

pResponse = A pointer to a TW_RGBRESPONSE structure.

Valid States

4 only

Description

Causes the Source to use its “identity” response curves for future RGB transfers. The identity curve causes no change in the values of the captured data when it is applied. (Note that resetting the curves for RGB data **does not** reset any MSG_SET curves for other pixel types).

Note: The DAT_RGBRESPONSE operations allow the application to specify the transfer curves that the Source should apply to the RGB data it acquires. The Source should not support these operations unless it can provide data of pixel type TWPT_RGB. The Source need not maintain actual “identity response curves” for use with the MSG_RESET operation—once reset, the Source should transfer the RGB data as acquired from the Source. The application should be sure that the Source supports these operations before invoking them. The operations should only be invoked when the active pixel type is RGB (TWPT_RGB). See Chapter 8 for information about the definitions and data structures used to describe the RGB response curve within TWAIN.

Application

No special action.

Source

Apply the identity response curve to all future RGB transfers. This means that the Source will transfer the RGB data exactly as acquired from the device.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL          /* Source does not support RGB  */
                             /* response curves              */
   TWCC_BADVALUE             /* Current pixel type is not    */
                             /* TWPT_RGB                     */
   TWCC_SEQERROR             /* Operation invoked in invalid */
                             /* state                        */
```

See Also

DG_IMAGE / DAT_RGBRESPONSE / MSG_SET

Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_RGBRESPONSE / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_RGBRESPONSE, MSG_SET, pResponse);
```

pResponse = A pointer to a TW_RGBRESPONSE structure.

Valid States

4 only

Description

Causes the Source to transform any RGB data according to the response curves specified by the application.

Application

Fill all three elements of the response curve with the response curve data you want the Source to apply to future RGB transfers. The application should consider writing the same values into each element of the same index to minimize color shift problems.

The Source may not support this operation. The application should ensure that the current pixel type is TWPT_RGB and examine the return code from this operation.

Source

Apply the specified response curves to all future RGB transfers.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL          /* Source does not support color */
                             /* response curves              */
   TWCC_BADVALUE             /* Current pixel type is not RGB */
   TWCC_SEQERROR             /* Operation invoked in invalid  */
                             /* state                        */
```

See Also

DG_IMAGE / DAT_RGBRESPONSE / MSG_RESET

Capability - ICAP_PIXELTYPE

8

Data Types and Data Structures

Chapter Contents

TWAIN defines a large number of data types and structures. These are all defined in the TWAIN.H file that is shipped as part of this toolkit. The file is written in C so you will need to modify the syntax if you develop your application or Source in some other language.

Naming Conventions

Data Structures, Variables, Pointers and Handles

Data structures referenced by pData parameter in DSM_Entry calls

Are prefixed by TW_ and followed by a descriptive name, in upper case. The name typically matches the call’s DAT parameter.

Example: TW_USERINTERFACE

Fields in data structures (not containing pointers or handles)

Typically, begin with a capital letter followed by mixed upper and lower case letters.

Example: The MinBufSize, MaxBufSize, and Preferred fields in which are in the TW_SETUPMEMXFER structure.

Fields in data structures that contain pointers or handles

Name starts with lower case “p” or “h” for pointer or handle followed by a typical field name with initial capital then mixed case characters.

Example: pData, hContainer

Constants and Types

General-use constants

Are prefixed by TWON_ followed by the description of the constant's meaning.

Example: TWON_ICONID, TWON_ARRAY

Specific-use constants

Are prefixed with TWxx_ where xx are two letters identifying the group to which the constant belongs.

Example: TWTY_INT16, TWTY_STR32 are constants of the group "TW Types"

Common data types

Rather than use the int, char, long, etc. types with their variations between compilers, TWAIN defines a group of types that are used to cast each data item used by the protocol. Types are prefixed and named exactly the same as TWAIN data structures, TW_ followed by a descriptive name, all in upper case characters.

Example: TW_UINT32, TW_HANDLE

Custom Constants

Applications and Sources may define their own private (custom) constant identifiers for any existing constant group by assigning the constant a value greater than or equal to 256. They may also define any new desired custom constant group. The consuming entity should check the originating entity's TW_IDENTITY.ProductName when encountering a constant value greater than or equal to 256 to see whether it can be recognized as a custom constant. Sources and applications should not assume that all entities will have such error checking built in, however.

The following are operation identifiers:

Data Groups	Prefixed with DG_
Data Argument Types	Prefixed with DAT_
Messages	Prefixed with MSG_
Return codes	Prefixed with TWRC_
Condition codes	Prefixed with TWCC_
General capabilities	Prefixed with CAP_
Image-specific capabilities	Prefixed with ICAP_
Audio-specific capabilities	Prefixed with ACAP_

As a general note, whenever the application or the Source allocates a TWAIN data structure, it should fill all the fields it is instructed to fill and write the default value (if one is specified) into any field it is not filling. If no default is specified, fill the field with the appropriate TWON_DONTCARExx constant where xx describes the size of the field in bits (bytes, in the case of strings). The TWON_ constants are described at the end of this chapter and defined in the TWAIN.H file.

Some fields return a value of -1 when the data to be returned is ambiguous or unknown. Applications and Sources must look for these special cases, especially when allocating memory. Examples of Fields with -1 values are found in TW_PENDINGXFERS (Count), TW_SETUPMEMXFER (MaxBufSize) and TW_IMAGEINFO (ImageWidth and ImageLength).

The remainder of this chapter lists the defined data types and data structures. Most of the constants are also listed. However, refer to the TWAIN.H file for more explanation about each constant and to see the lengthy list of country constants which are not duplicated here.

Platform Dependent Definitions and Typedefs

On Windows

```
typedef HANDLE            TW_HANDLE;
typedef LPVOID            TW_MEMREF;
```

On Macintosh

```
#define PASCAL            pascal
#define FAR
typedef Handle            TW_HANDLE;
typedef char              *TW_MEMREF;
```

On UNIX

```
#define PASCAL            pascal
typedef unsigned char     *TW_HANDLE;
typedef unsigned char     *TW_MEMREF;
```

Definitions of Common Types

String types

```
typedef char     TW_STR32[34],       FAR *pTW_STR32;
typedef char     TW_STR64[66],       FAR *pTW_STR64;
typedef char     TW_STR128[130],     FAR *pTW_STR128;
typedef char     TW_STR255[256],     FAR *pTW_STR255;
```

On Windows: These include room for the strings and a NULL character.

On Macintosh: These include room for a length byte followed by the string.

Note: he TW_STR255 must hold less than 256 characters so the length fits in the first byte on Macintosh.

Numeric types

```
typedef char              TW_INT8      FAR *pTW_INT8;
typedef short             TW_INT16     FAR *pTW_INT16;
typedef long              TW_INT32     FAR *pTW_INT32;
typedef unsigned char     TW_UINT8     FAR *pTW_UINT8;
typedef unsigned short    TW_UINT16    FAR *pTW_UINT16;
typedef unsigned long     TW_UINT32    FAR *pTW_UINT32;
typedef unsigned short    TW_BOOL      FAR *pTW_BOOL;
```

Fixed point structure type

```
typedef struct {
        TW_INT16        Whole;
        TW_UINT16       Frac;
} TW_FIX32,    FAR *pTW_FIX32;
```

Note: In cases where the data type is smaller than TW_UINT32, the data should reside in the lower word.

Data Structure Definitions

This section provides descriptions of the data structure definitions.

TW_ARRAY

```
typedef struct {
   TW_UINT16       ItemType;
   TW_UINT32       NumItems;
   TW_UINT8        ItemList[1];
} TW_ARRAY, FAR * pTW_ARRAY;
```

Used by

TW_CAPABILITY structure (when ConType field specifies TWON_ARRAY)

Description

This structure stores a group of associated individual values which, when taken as a whole, describes a single "value" for a capability. The values need have no relationship to one another aside from being used to describe the same "value" of the capability. Such an array of values is useful to describe the CAP_SUPPORTEDCAPS list. This structure is used as a member of TW_CAPABILITY structures. Since this structure does not, therefore, exist "stand-alone" it is identified by a TWON_xxxx constant rather than a DAT_xxxx. This structure is related in function and purpose to TW_ENUMERATION, TW_ONEVALUE, and TW_RANGE.

Field Descriptions

ItemType	The type of items in the array. The type is indicated by the constant held in this field. The constant is of the kind TWTY_xxxx. All items in the array have the same size.
NumItems	How many items are in the array.
ItemList[1]	This is the array. One value resides within each element of the array. Space for the array is not allocated inside this structure. The ItemList value is simply a placeholder for the start of the actual array, which must be allocated when the container is allocated . Remember to typecast the allocated array, as well as references to the elements of the array, to the type indicated by the constant in ItemType.

TW_AUDIOINFO

```
typedef struct {
   TW_STR255      Name;
   TW_UINT32      Reserved;
} TW_AUDIOINFO, FAR * pTW_AUDIOINFO;
```

Used by

The DG_AUDIO / DAT_AUDIOINFO / MSG_GET operation

Description

Field Descriptions

Name	Name of audio data
Reserved	Reserved space

TW_CAPABILITY

```
typedef struct {
   TW_UINT16       Cap;
   TW_UINT16       ConType;
   TW_HANDLE       hContainer;
} TW_CAPABILITY, FAR * pTW_CAPABILITY;
```

Used by

DG_CONTROL / DAT_CAPABILITY / MSG_GET
DG_CONTROL / DAT_CAPABILITY / MSG_GETCURRENT
DG_CONTROL / DAT_CAPABILITY / MSG_GETDEFAULT
DG_CONTROL / DAT_CAPABILITY / MSG_RESET
DG_CONTROL / DAT_CAPABILITY / MSG_SET

Description

Used by an application either to get information about, or control the setting of a capability. The first field identifies the capability being negotiated (e.g., ICAP_BRIGHTNESS). The second specifies the format of the container (e.g., TWON_ONEVALUE). The third is a handle (HGLOBAL under Microsoft Windows) to the container itself.

The application always sets the Cap field. On MSG_SET, the application also sets the ConType and hContainer fields. On MSG_RESET, MSG_GET, MSG_GETCURRENT, and MSG_GETDEFAULT, the source fills in the ConType and hContainer fields.

It is always the application's responsibility to free the container when it is no longer needed. On a MSG_GET, MSG_GETCURRENT, or MSG_GETDEFAULT, the source allocates the container but ownership passes to the application. On a MSG_SET, the application provides the container either by allocating it or by re-using a container created earlier.

On a MSG_SET, the Source must not modify the container and it must copy any data that it wishes to retain.

Field Descriptions

Cap	The numeric designator of the capability (of the form CAP_xxxx, ICAP_xxxx, or ACAP_xxxx). e.g. ICAP_BRIGHTNESS. A list of these can be found in Chapter 9 and in the TWAIN.H file.
ConType	The type of the container referenced by hContainer. The container structure will be one of four types: TWON_ARRAY, TWON_ENUMERATION, TWON_ONEVALUE, or TWON_RANGE. One of these values, which types the container, should be entered into this field by whichever TWAIN entity fills in the container. When the application wants to set (MSG_SET) the Source's capability, the application must fill in this field. When the application wants to get (MSG_GET) capability information from the Source, the Source must fill in this field.
hContainer	References the container structure where detailed information about the capability is stored. When the application wants to set (MSG_SET) the Source's capability, the application must provide the hContainer. When the application wants to get (MSG_GET) the Source's capability information, the Source must allocate the space for the container. In either case, the application must release this space.

TW_CIECOLOR

```
typedef struct {
   TW_UINT16          ColorSpace
   TW_INT16           LowEndian;
   TW_INT16           DeviceDependent;
   TW_INT32           VersionNumber;
   TW_TRANSFORMSTAGE  StageABC;
   TW_TRANSFORMSTAGE  StageLMN;
   TW_CIEPOINT        WhitePoint;
   TW_CIEPOINT        BlackPoint;
   TW_CIEPOINT        WhitePaper;
   TW_CIEPOINT        BlackInk;
   TW_FIX32           Samples[1];
} TW_CIECOLOR, FAR *  pTW_CIECOLOR;
```

Used by

DG_IMAGE / DAT_CIECOLOR / MSG_GET

Description

Defines the mapping from an RGB color space device into CIE 1931 (XYZ) color space. For more in-depth information, please reference the PostScript Language Reference Manual, Second Edition, pp. 173-193. Note that the field names do not follow the conventions used elsewhere within TWAIN. This breach allows the identifiers shown here to exactly match those described in Appendix A, which was not written specifically for this Toolkit. Please also note that ColorSpace has been redefined from its form in Appendix A to use TWPT_xxxx constants defined in the TWAIN.H file.

This structure closely parallels the TCIEBasedColorSpace structure definition in Appendix A. Note that the field names are slightly different and that two new fields have been added (WhitePaper and BlackInk) to describe the reflective characteristics of the page from which the image was acquired.

If the Source can provide TWPT_CIEXYZ, it must support all operations on this structure.

Field Descriptions

ColorSpace	Defines the original color space that was transformed into CIE XYZ. Use a constant of type TWPT_xxxx. This value is not set-able by the application. Application should write TWON_DONTCARE16 into this on a MSG_SET.
LowEndian	Used to indicate which data byte is taken first. If zero, then high byte is first. If non-zero, then low byte is first.
DeviceDependent	If non-zero then color data is device-dependent and only ColorSpace is valid in this structure.
VersionNumber	Version of the color space descriptor specification used to define the transform data. The current version is zero.
StageABC	Describes parametrics for the first stage transformation of the Postscript Level 2 CIE color space transform process.
StageLMN	Describes parametrics for the first stage transformation of the Postscript Level 2 CIE color space transform process.
WhitePoint	Values that specify the CIE 1931 (XYZ space) tri-stimulus value of the diffused white point.
BlackPoint	Values that specify the CIE 1931 (XYZ space) tri-stimulus value of the diffused black point.
WhitePaper	Values that specify the CIE 1931 (XYZ space) tri-stimulus value of ink-less “paper” from which the image was acquired.
BlackInk	Values that specify the CIE 1931 (XYZ space) tri-stimulus value of solid black ink on the “paper” from which the image was acquired.
Samples[1]	Optional table look-up values used by the decode function. Samples are ordered sequentially and end-to-end as A, B, C, L, M, and N.

TW_CIEPOINT

```
typedef struct {
   TW_FIX32       X;
   TW_FIX32       Y;
   TW_FIX32       Z;
} TW_CIEPOINT, FAR * pTW_CIEPOINT;
```

Used by

Embedded in the TW_CIECOLOR structure

Description

Defines a CIE XYZ space tri-stimulus value. This structure parallels the TCIEPoint structure definition in Appendix A.

Field Descriptions

X	First tri-stimulus value of the CIE space representation.
Y	Second tri-stimulus value of the CIE space representation.
Z	Third tri-stimulus value of the CIE space representation.

TW_CUSTOMDSDATA

```
typedef struct {
   TW_UINT32        InfoLength;      /* Length (in bytes) of data */
   TW_UINT8         InfoData[1];     /* Array (Length) bytes long */
} TW_CUSTOMDSDATA, FAR *pTW_CUSTOMDSDATA;
```

Used by

DG_CONTROL / DAT_CUSTOMDSDATA / MSG_GET
DG_CONTROL / DAT_CUSTOMDSDATA / MSG_SET

Description

Allows for a data source and application to pass custom data to each other.

The format of the data contained in InfoData will be data source specific and will not be defined by the TWAIN API. This structure will be used by an application to query the data source for it's current settings, and to archive them to disk. Although the format for this custom data is not defined by TWAIN, source implementers are encouraged to use a ASCII representation for the custom data to be used for settings archival. A Windows INI style format would be easy to implement and allow for additional features to be added without breaking backwards compatibility.

It is also recommended that source vendors embed basic source revision and vendor ID information in the InfoData body so they can determine if the structure being based to the data source is correct.

Field Descriptions

InfoLength	Length, in bytes, of data
InfoData[1]	Array (length) bytes long

TW_DECODEFUNCTION

```
typedef struct {
   TW_FIX32       StartIn;
   TW_FIX32       BreakIn;
   TW_FIX32       EndIn;
   TW_FIX32       StartOut;
   TW_FIX32       BreakOut;
   TW_FIX32       EndOut;
   TW_FIX32       Gamma;
   TW_FIX32       SampleCount;
} TW_DECODEFUNCTION, FAR * pTW_DECODEFUNCTION;
```

Used by

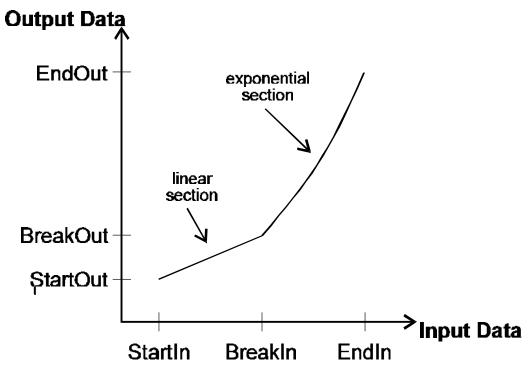
Embedded in the TW_TRANSFORMSTAGE structure that is embedded in the TW_CIECOLOR structure

Description

Defines the parameters used for channel-specific transformation. The transform can be described either as an extended form of the gamma function or as a table look-up with linear interpolation. This structure parallels the TDecodeFunction structure definition in Appendix A.

Field Descriptions

StartIn	Starting input value of the extended gamma function. Defines the minimum input value of channel data.
BreakIn	Ending input value of the extended gamma function. Defines the maximum input value of channel data.
EndIn	The input value at which the transform switches from linear transformation/interpolation to gamma transformation.
StartOut	Starting output value of the extended gamma function. Defines the minimum output value of channel data.
BreakOut	Ending output value of the extended gamma function. Defines the maximum output value of channel data.
EndOut	The output value at which the transform switches from linear transformation/interpolation to gamma transformation.
Gamma	Constant value. The exponential used in the gamma function.
SampleCount	The number of samples in the look-up table. Includes the values of StartIn and EndIn. Zero-based index (actually, number of samples - 1). If zero, use extended gamma, otherwise use table look-up.



Extended Gamma Parameters

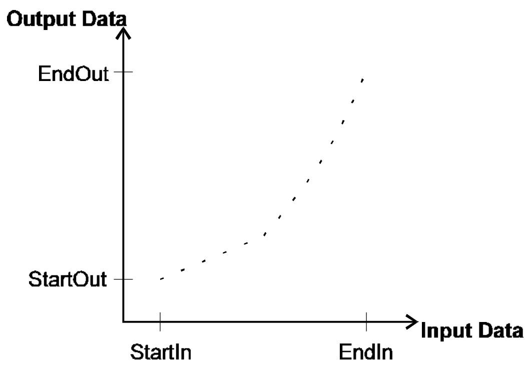


Table Look-up Parameters

TW_DEVICEEVENT

```
typedef struct {
   TW_UINT32      Event;
   TW_STR255      DeviceName;
   TW_UINT32      BatteryMinutes;         // Battery Minutes Remaining
   TW_INT16       BatteryPercentage;      // Battery Percentage Remaining
   TW_INT32       PowerSupply;            // Power Supply
   TW_FIX32       XResolution;            // Resolution
   TW_FIX32       YResolution;            // Resolution
   TW_UINT32      FlashUsed2;             // Flash Used2
   TW_UINT32      AutomaticCapture;       // Automatic Capture
   TW_UINT32      TimeBeforeFirstCapture; // Automatic Capture
   TW_UINT32      TimeBetweenCaptures;    // Automatic Capture
} TW_DEVICEEVENT, FAR * pTW_DEVICEEVENT;
```

Used by

DG_CONTROL / DAT_DEVICEEVENT / MSG_GET

Description

Provides information about the Event that was raised by the Source. The Source should only fill in those fields applicable to the Event. The Application must only read those fields applicable to the Event.

Field Descriptions

Event	One of the TWDE_xxxx values. Defines the event that has taken place.
DeviceName	The name of the device that generated the event.
Valid for TWDE_BATTERYCHECK only	
BatteryMinutes	Minutes of battery power remaining.
BatteryPercentage	Percentage of battery power remaining.
Valid for TWDE_POWERSUPPLY only	
PowerSupply	Current power supply in use.
Valid for TWDE_RESOLUTION only	
XResolution	Current X Resolution.
YResolution	Current Y Resolution.
Valid for TWDE_FLASHUSED2 only	
FlashUsed2	Current flash setting.
Valid for TWDE_AUTOMATICCAPTURE only	
AutomaticCapture	Number of images camera will capture.
TimeBeforeFirstCapture	Number of seconds before first capture.
TimeBetweenCaptures	Hundredths of a second between captures.

TW_ELEMENT8

```
typedef struct {
   TW_UINT8       Index;
   TW_UINT8       Channel1;
   TW_UINT8       Channel2;
   TW_UINT8       Channel3;
 } TW_ELEMENT8, FAR * pTW_ELEMENT8;
```

Used by

Embedded in the TW_GRAYRESPONSE, TW_PALETTE8 and TW_RGBRESPONSE structures

Description

This structure holds the tri-stimulus color palette information for TW_PALETTE8 structures. The order of the channels shall match their alphabetic representation. That is, for RGB data, R shall be channel 1. For CMY data, C shall be channel 1. This allows the application and Source to maintain consistency. Grayscale data will have the same values entered in all three channels.

Field Descriptions

Index	Value used to index into the color table. Especially useful on the Macintosh.
Channel1	First tri-stimulus value (e.g. Red).
Channel2	Second tri-stimulus value (e.g. Green).
Channel3	Third tri-stimulus value (e.g. Blue).

TW_ENUMERATION

```
typedef struct {
  TW_UINT16      ItemType;
  TW_UINT32      NumItems;
  TW_UINT32      CurrentIndex;
  TW_UINT32      DefaultIndex;
  TW_UINT8       ItemList[1];
} TW_ENUMERATION, FAR * pTW_ENUMERATION;
```

Used by

TW_CAPABILITY structure (when ConType field specifies TWON_ENUMERATION)

Description

Stores a group of individual values describing a capability. The values are ordered from lowest to highest values, but the step size between each value is probably not uniform. Such a list would be useful to describe the discreet resolutions of a capture device supporting, say, 75, 150, 300, 400, and 800 dots per inch.

This structure is related in function and purpose to TW_ARRAY, TW_ONEVALUE, and TW_RANGE.

Field Descriptions

ItemType	The type of items in the enumerated list. The type is indicated by the constant held in this field. The constant is of the kind TWTY_xxxx. All items in the array have the same size.
NumItems	How many items are in the enumeration.
CurrentIndex	The item number, or index (zero-based) into ItemList[], of the “current” value for the capability.
DefaultIndex	The item number, or index (zero-based) into ItemList[], of the “power-on” value for the capability.
ItemList[1]	The enumerated list: one value resides within each array element. Space for the list is not allocated inside this structure. The ItemList value is simply a placeholder for the start of the actual array, which must be allocated when the container is allocated. Remember to typecast the allocation to ItemType, as well as references to the elements of the array.

TW_EVENT

```
typedef struct {
   TW_MEMREF      pEvent;
   TW_UINT16      TWMessage;
TW_EVENT, FAR * pTW_EVENT;
```

Used by

DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT

Description

Used to pass application events / messages from the application to the Source. The Source is responsible for examining the event / message, deciding if it belongs to the Source, and returning an appropriate return code to indicate whether or not the Source owns the event / message. This process is covered in more detail in the Event Loop section of Chapter 3.

Field Descriptions

pEvent	A pointer to the event / message to be examined by the Source. Under Microsoft Windows, pEvent is a pMSG (pointer to a Microsoft Windows MSG struct). That is, the message the application received from GetMessage(). On the Macintosh, pEvent is a pointer to an EventRecord.
TWMessage	Any message (MSG_xxxx) the Source needs to send to the application in response to processing the event / message. The messages currently defined for this purpose are MSG_NULL, MSG_XFERREADY and MSG_CLOSEDSREQ.

TW_EXTIMAGEINFO

```
typedef struct {
      TW_UINT32        NumInfos;
      TW_INFO          Info[1];
} TW_EXTIMAGEINFO, FAR * pTW_ EXTIMAGEINFO;
```

Used by

DG_IMAGE / DAT_EXTIMAGEINFO / MSG_GET

Description

This structure is used to pass extended image information from the data source to application at the end of State 7. The application creates this structure at the end of State 7, when it receives XFERDONE. Application fills NumInfos for Number information it needs, and array of extended information attributes in Infos[ ] array. Application, then, sends it down to the source using the above operation triplet. The data source then examines each Info, and fills the rest of data with information allocating memory when necessary.

Field Descriptions

NumInfos	Number of information that application is requesting. This is filled by the application. If positive, then the application is requesting specific extended image information. The application should allocate memory and fill in the attribute tag for image information.
Info[1]	Array of information. See TW_INFO structure.

TW_FILESYSTEM

```
typedef struct {
   // DG_CONTROL / DAT_FILESYSTEM / MSG_xxxx fields…
   TW_STR255    InputName;
   TW_STR255    OutputName;
   TW_MEMREF    Context;
   // DG_CONTROL / DAT_FILESYSTEM / MSG_COPY
   // DG_CONTROL / DAT_FILESYSTEM / MSG_DELETE field…
   int          Recursive;
   // DG_CONTROL / DAT_FILESYSTEM / MSG_GETINFO fields…
   TW_INT32     FileType;
   TW_UINT32    Size;
   TW_STR32     CreateTimeDate;
   TW_STR32     ModifiedTimeDate;
   TW_UINT32    FreeSpace;
   TW_INT32     NewImageSize;
   TW_UINT32    NumberOfFiles;
   TW_UINT32    NumberOfSnippets;
   TW_UINT32    DeviceGroupMask;
   char         Reserved[512];
} TW_FILESYSTEM, FAR * pTW_FILESYSTEM;
```

Used by

DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_COPY
DG_CONTROL / DAT_FILESYSTEM / MSG_CREATEDIRECTORY
DG_CONTROL / DAT_FILESYSTEM / MSG_DELETE
DG_CONTROL / DAT_FILESYSTEM / MSG_FORMATMEDIA
DG_CONTROL / DAT_FILESYSTEM / MSG_GETCLOSE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETFIRSTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_GETINFO
DG_CONTROL / DAT_FILESYSTEM / MSG_GETNEXTFILE
DG_CONTROL / DAT_FILESYSTEM / MSG_RENAME

Description

Provides information about the currently selected device.

Field Descriptions

InputName	The name of the input or source file.
OutputName	The result of an operation or the name of a destination file.
Context	A pointer to Source specific data used to remember state information, such as the current directory.

MSG_GETINFO / MSG_GETFILEFIRST / MSG_DELETE

Recursive	When set to TRUE recursively apply the operation. (ex: deletes all subdirectories in the directory being deleted; or copies all sub-directories in the directory being copied.

MSG_GETINFO / MSG_GETFILEFIRST / MSG_GETFILENEXT

FileType	One of the TWFS_xxxx values.
Size	TWFT_DIRECTORY - Total size of media in bytes. TWFT_IMAGE - Size of image in bytes. TWFT_xxxx - All other file types return a value of 0.
CreateTimeDate	The create date of the file, in the form "YYYY/MM/DD HH:mm:SS:sss" where YYYY is the year, MM is the numerical month, DD is the numerical day, HH is the hour, mm is the minute, SS is the second, and sss is the millisecond.
ModifyTimeDate	Last date the file was modified. Same format as *CreateTimeDate*.
FreeSpace	The bytes of free space left on the current device.
NewImageSize	An estimate of the amount of space a new image would take up, based on image layout, resolution and compression. Dividing this value into the *FreeSpace* will yield the approximate number of images that the Device has room for.
NumberOfFiles	TWFT_IMAGE - Return 0 TWFT_xxxx - Return number of TWFT_IMAGE files on the file system including those in all sub-directories.
NumberOfSnippets	The number of audio snippets associated with a file of type TWFY_IMAGE.
DeviceGroupMask	A set of bits, with each bit uniquely identifying a device of type TWFY_CAMERA and any associated TWFY_CAMERATOP and/or TWFY_CAMERABOTTOM devices. See the article on File Systems in Appendix A of this specification for more information.
Reserved	Space reserved for future expansion of this structure.

TW_FIX32

```
typedef struct {
   TW_INT16       Whole;
   TW_UINT16      Frac;
} TW_FIX32, FAR * pTW_FIX32;
```

Used by

Embedded in the TW_CIECOLOR, TW_CIEPOINT, TW_DECODEFUNCTION, TW_FRAME, TW_IMAGEINFO, and TW_TRANSFORMSTAGE structures.

Used in TW_ARRAY, TW_ENUMERATION, TW_ONEVALUE, and TW_RANGE structures when ItemType is TWTY_FIX32.

Description

Stores a Fixed point number in two parts, a whole and a fractional part. The Whole part carries the sign for the number. The Fractional part is unsigned.

Field Descriptions

Whole	The Whole part of the floating point number. This number is signed.
Frac	The Fractional part of the floating point number. This number is unsigned.

The following functions convert TW_FIX32 to float and float to TW_FIX32:

```
/**********************************************************
* FloatToFix32
* Convert a floating point value into a FIX32.
**********************************************************/

TW_FIX32 FloatToFix32 (float floater)
{
   TW_FIX32 Fix32_value;
   TW_INT32 value = (TW_INT32) (floater * 65536.0 + 0.5);
   Fix32_value.Whole = value >> 16;
   Fix32_value.Frac = value & 0x0000ffffL;
   return (Fix32_value);
}
/**********************************************************
* Fix32ToFloat
* Convert a FIX32 value into a floating point value.
**********************************************************/

float FIX32ToFloat (TW_FIX32 fix32)
{
   float     floater;
   floater = (float) fix32.Whole + (float) fix32.Frac / 65536.0;

   return floater;
}
```

TW_FRAME

```
typedef struct {
   TW_FIX32      Left;
   TW_FIX32      Top;
   TW_FIX32      Right;
   TW_FIX32      Bottom;
} TW_FRAME, FAR * pTW_FRAME;
```

Used by

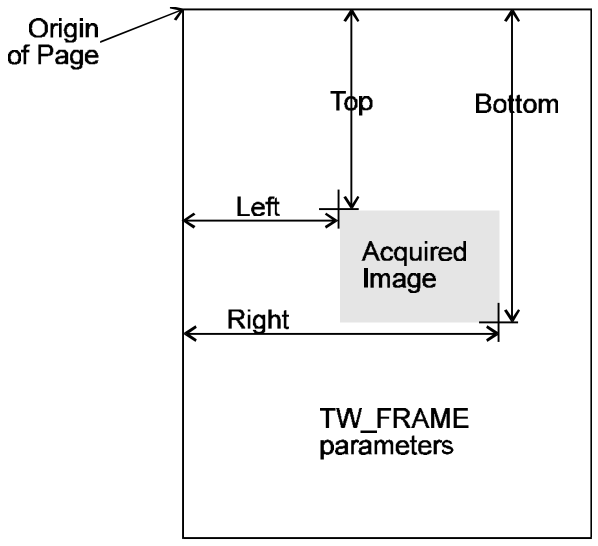
Embedded in the TW_IMAGELAYOUT structure

Description

Defines a frame rectangle in ICAP_UNITS coordinates.

Field Descriptions

Left	Value of the left-most edge of the rectangle (in ICAP_UNITS).
Top	Value of the top-most edge of the rectangle (in ICAP_UNITS).
Right	Value of the right-most edge of the rectangle (in ICAP_UNITS).
Bottom	Value of the bottom-most edge of the rectangle (in ICAP_UNITS).



Frame Parameters

TW_GRAYRESPONSE

```
typedef struct {
   TW_ELEMENT8       Response[1];
} TW_GRAYRESPONSE, FAR * pTW_GRAYRESPONSE;
```

Used by

DG_IMAGE / DAT_GRAYRESPONSE / MSG_RESET
DG_IMAGE / DAT_GRAYRESPONSE / MSG_SET

Description

This structure is used by the application to specify a set of mapping values to be applied to grayscale data. Use this structure for grayscale data whose bit depth is up to and including 8-bits. This structure can only be used if TW_IMAGEINFO.PixelType is TWPT_GRAY. The number of elements in the array is determined by TW_IMAGEINFO.BitsPerPixel—the number of elements is 2 raised to the power of TW_IMAGEINFO.BitsPerPixel.

This structure is primarily intended for use by applications that bypass the Source's built-in user interface.

Field Descriptions

Response[1]	Transfer curve descriptors. All three channels must contain the same value for every entry.

TW_HANDLE

On Windows:

```
typedef HANDLE              TW_HANDLE;
```

On Macintosh:

```
typedef Handle              TW_HANDLE;
```

On Unix:

```
typedef unsigned char      *TW_HANDLE;
```

Used by

Embedded in the TW_CAPABILITY and TW_USERINTERFACE structures

Description

The typedef of Handles are defined by the operating system. TWAIN defines TW_HANDLE to be the handle type supported by the operating system.

Field Descriptions

See definitions above

TW_IDENTITY

```
typedef struct {
   TW_UINT32       Id;
   TW_VERSION      Version;
   TW_UINT16       ProtocolMajor;
   TW_UINT16       ProtocolMinor;
   TW_UINT32       SupportedGroups;
   TW_STR32        Manufacturer;
   TW_STR32        ProductFamily;
   TW_STR32        ProductName;
} TW_IDENTITY, FAR * pTW_IDENTITY;
```

Used by

A large number of the operations because it identifies the application and the Source

Description

Provides identification information about a TWAIN entity. Used to maintain consistent communication between entities.

Field Descriptions

Id	A unique, internal identifier for the TWAIN entity. This field is only filled by the Source Manager. Neither an application nor a Source should fill this field. The Source uses the contents of this field to “identify” which application is invoking the operation sent to the Source.
Version	A TW_VERSION structure identifying the TWAIN entity.
ProtocolMajor	Major number of latest TWAIN version that this element supports (see TWON_PROTOCOLMAJOR).
ProtocolMinor	Minor number of latest TWAIN version that this element supports (see TWON_PROTOCOLMINOR).
SupportedGroups	1. The application will normally set this field to specify which Data Group(s) it wants the Source Manager to sort Sources by when presenting the Select Source dialog, or returning a list of available Sources. The application sets this prior to invoking a MSG_USERSELECT operation. 2. The application may also set this field to specify which Data Group(s) it wants the Source to be able to acquire and transfer. The application must do this prior to sending the Source its MSG_ENABLEDS operation. 3. The Source must set this field to specify which Data Group(s) it can acquire. It will do this in response to a MSG_OPENDS.
Manufacturer	String identifying the manufacturer of the application or Source. e.g. “Aldus”.
ProductFamily	Tells an application that performs device-specific operations which product family the Source supports. This is useful when a new Source has been released and the application doesn’t know about the particular Source but still wants to perform Custom operations with it. e.g. “ScanMan”.
ProductName	A string uniquely identifying the Source. This is the string that will be displayed to the user at Source select-time. This string must uniquely identify your Source for the user, and should identify the application unambiguously for Sources that care. e.g. “ScanJet IIc”.

TW_IMAGEINFO

```
typedef struct {
   TW_FIX32       XResolution;
   TW_FIX32       YResolution;
   TW_INT32       ImageWidth;
   TW_INT32       ImageLength;
   TW_INT16       SamplesPerPixel;
   TW_INT16       BitsPerSample[8];
   TW_INT16       BitsPerPixel;
   TW_BOOL        Planar;
   TW_INT16       PixelType;
   TW_UINT16      Compression;
} TW_IMAGEINFO, FAR * pTW_IMAGEINFO;
```

Used by

The DG_IMAGE / DAT_IMAGEINFO / MSG_GET operation

Description

Describes the "real" image data, that is, the complete image being transferred between the Source and application. The Source may transfer the data in a different format--the information may be transferred in "strips" or "tiles" in either compressed or uncompressed form. See the TW_IMAGEMEMXFER structure for more information.

The term "sample" is referred to a number of times in this structure. It holds the same meaning as in the TIFF specification. A sample is a contiguous body of image data that can be categorized by the channel or "ink color" it was captured to describe. In an R-G-B (Red-Green-Blue) image, such as on your TV or computer's CRT, each color channel is composed of a specific color. There are 3 samples in an R-G-B; Red, Green, and Blue. A C-Y-M-K image has 4 samples. A Grayscale or Black and White image has a single sample.

Note: The value -1 in ImageWidth and ImageLength are special cases. It is possible for a Source to not know either its Width or Length. Applications need to consider this when allocating memory or otherwise dealing with the size of the Image.

Field Descriptions

XResolution	The number of pixels per ICAP_UNITS in the horizontal direction. The current unit is assumed to be “inches” unless it has been otherwise negotiated between the application and Source.
YResolution	The number of pixels per ICAP_UNITS in the vertical direction.
ImageWidth	How wide, <u>in pixels</u>, the entire image to be transferred is. If the Source doesn’t know, set this field to -1 (hand scanners may do this). --1 can only be used if the application has set ICAP_UNDEFINEDIMAGESIZE to TRUE.
ImageLength	How tall/long, <u>in pixels</u>, the image to be transferred is. If the Source doesn’t know, set this field to -1 (hand scanners may do this). -1 can only be used if the application has set ICAP_UNDEFINEDIMAGESIZE to TRUE.
SamplesPerPixel	The number of samples being returned. For R-G-B, this field would be set to 3. For C-M-Y-K, 4. For Grayscale or Black and White, 1.
BitsPerSample[8]	For each sample, the number of bits of information. 24-bit R-G-B will typically have 8 bits of information in each sample (8+8+8). Some 8-bit color is sampled at 3 bits Red, 3 bits Green, and 2 bits Blue. Such a scheme would put 3, 3, and 2 into the first 3 elements of this array. The supplied array allows up to 8 samples. Samples are not limited to 8 bits. However, both the application and Source must simultaneously support sample sizes greater than 8 bits per color.
BitsPerPixel	The number of bits in each image pixel (or bit depth). This value is invariant across the image. 24-bit R-G-B has BitsPerPixel = 24. 40-bit C-M-Y-K has BitsPerPixel=40. 8-bit Grayscale has BitsPerPixel = 8. Black and White has BitsPerPixel = 1.
Planar	If SamplesPerPixel > 1, indicates whether the samples follow one another on a pixel-by-pixel basis (R-G-B-R-G-B-R-G-B...) as is common with a one-pass scanner or all the pixels for each sample are grouped together (complete group of R, complete group of G, complete group of B) as is common with a three-pass scanner. If the pixel-by-pixel method (also known as “chunky”) is used, the Source should set Planar = FALSE. If the grouped method (also called “planar”) is used, the Source should set Planar = TRUE.
PixelType	This is the highest categorization for how the data being transferred should be interpreted by the application. This is how the application can tell if the data is Black and White, Grayscale, or Color. Currently, the only color type defined is “tri-stimulus”, or color described by three characteristics. Most popular color description methods use tri-stimulus descriptors. For simplicity, the constant used to identify tri-stimulus color is called TWPT_RBG, for R-G-B color. There is no default for this value. Fill this field with the appropriate TWPT_xxxx constant.
Compression	The compression method used to process the data being transferred. Default is no compression. Fill this field with the appropriate TWCP_xxxx constant.

TW_IMAGELAYOUT

```
typedef struct {
   TW_FRAME       Frame;
   TW_UINT32      DocumentNumber;
   TW_UINT32      PageNumber;
   TW_UINT32      FrameNumber;
} TW_IMAGELAYOUT, FAR * pTW_IMAGELAYOUT;
```

Used by

DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT
DG_IMAGE / DAT_IMAGELAYOUT / MSG_RESET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET

Description

Involves information about the original size of the acquired image and its position on the original "page" relative to the "page's" upper-left corner. **Default measurements are in inches** (units of measure can be changed by negotiating the ICAP_UNITS capability). This information may be used by the application to relate the acquired (and perhaps processed image) to the original. Further, the application can, using this structure, set the size of the image it wants acquired.

Another attribute of this structure is the included frame, page, and document indexing information. Most Sources and applications, at least at first, will likely set all these fields to one. For Sources that can acquire more than one frame from a page in a single acquisition, the FrameNumber field will be handy. Sources that can acquire more than one page from a document feeder will use PageNumber and DocumentNumber. These fields will be especially useful for forms-processing applications and other applications with similar document tracking requirements.

Field Descriptions

Frame	Defines the Left, Top, Right, and Bottom coordinates (in ICAP_UNITS) of the rectangle enclosing the original image on the original “page”. If the application isn’t interested in setting the origin of the image, set both Top and Left to zero. The Source will fill in the actual values following the acquisition. See also TW_FRAME.
DocumentNumber	The document number, assigned by the Source, that the acquired data originated on. Useful for grouping pages together. Usually a physical representation, this could just as well be a logical construct. Initial value is 1. Increment when a new document is placed into the document feeder (usually tell this has happened when the feeder empties). Reset when no longer acquiring from the feeder.
PageNumber	The page which the acquired data was captured from. Useful for grouping Frames together that are in some way related, usually Source. Usually a physical representation, this could just as well be a logical construct. Initial value is 1. Increment for each page fed from a page feeder. Reset when a new document is placed into the feeder.
FrameNumber	Usually a chronological index of the acquired frame. These frames are related to one another in some way; usually they were acquired from the same page. The Source assigns these values. Initial value is 1. Reset when a new page is acquired from.

TW_IMAGEMEMXFER

```
typedef struct {
   TW_UINT16       Compression;
   TW_UINT32       BytesPerRow;
   TW_UINT32       Columns;
   TW_UINT32       Rows;
   TW_UINT32       XOffset;
   TW_UINT32       YOffset;
   TW_UINT32       BytesWritten;
   TW_MEMORY       Memory;
} TW_IMAGEMEMXFER, FAR * pTW_IMAGEMEMXFER;
```

Used by

DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET

Description

Describes the form of the acquired data being passed from the Source to the application. When used in combination with a TW_IMAGEINFO structure, the application can correctly interpret the image.

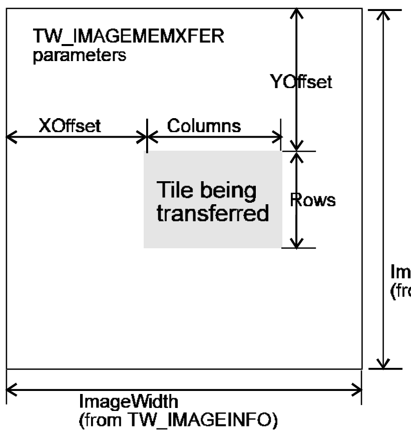
This structure allows transfer of “chunks” from the acquired data. These portions may be either “strips” or “tiles.” Strips are tiles whose width matches that of the full image. Strips are always passed sequentially, from “top” to “bottom”. A tile’s position does not necessarily follow that of the previously passed tile. Most Sources will transfer strips.

Note: The application should remember what corner was contained in the first tile of a plane. When the opposite corner is delivered, the plane is complete. The dimensions of the memory transfers may vary.

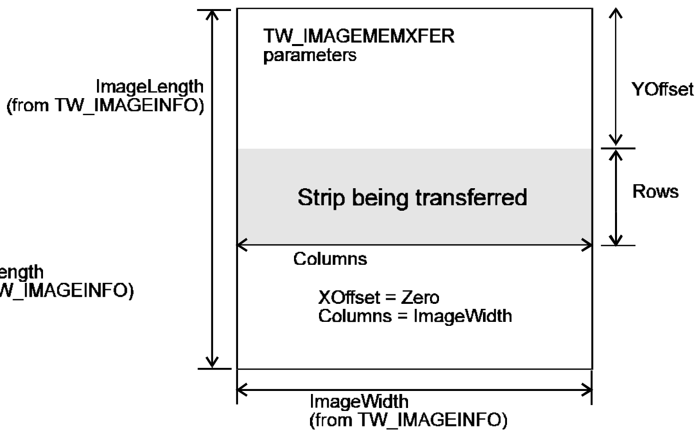
Data may be passed either compressed or uncompressed. All Sources must pass uncompressed data. Sources are not required to support compressed data transfers. Compressed data transfers, and how the values are entered into the fields of this structure, are described in Chapter 4.

Following is a picture of some of the fields from a TW_IMAGEMEMXFER structure. **The large outline shows the entire image which was selected to be transferred.** The smaller rectangle shows the particular portion being described by this TW_IMAGEMEMXFER structure.

Note: Remember that for a “strip” transfer XOffset = 0, and Columns = TW_IMAGEINFO.ImageWidth.


TW_IMAGEMEMXFER parameters
YOffset
XOffset
Columns
Tile being transferred
Rows
ImageLength (from TW_IMAGEINFO)
ImageWidth (from TW_IMAGEINFO)

Tile Positioning


TW_IMAGEMEMXFER parameters
ImageLength (from TW_IMAGEINFO)
YOffset
Strip being transferred
Rows
Columns
XOffset = Zero
Columns = ImageWidth
ImageWidth (from TW_IMAGEINFO)

Strip Positioning

Field Descriptions

Compression	The compression method used to process the data being transferred. Write the constant (TWCP_xxxx) that precisely describes the type of compression used for the buffer. This may be different from the method reported in the TW_IMAGEINFO structure (if the user selected a different method before the actual transfer began, for instance). This is unlikely, but possible. The application can optionally abort the acquisition if the value in this field differs from the TW_IMAGEINFO value. Default is no compression (TWCP_NONE) and most transfers will probably be uncompressed. See the list of constants in the TWAIN.H file.
BytesPerRow	The number of uncompressed bytes in each row of the piece of the image being described in this buffer.
Columns	The number of uncompressed columns (in pixels) in this buffer.
Rows	The number or uncompressed rows (in pixels) in this buffer.
XOffset	How far, in pixels, the left edge of the piece of the image being described by this structure is inset from the “left” side of the original image. If the Source is transferring in “strips”, this value will equal zero. If the Source is transferring in “tiles”, this value will often be non-zero.
YOffset	Same idea as XOffset, but the measure is in pixels from the “top” of the original image to the upper edge of this piece.
BytesWritten	The number of bytes written into the transfer buffer. This field must always be filled in correctly, whether compressed or uncompressed data is being transferred.
Memory	A structure of type TW_MEMORY describing who must dispose of the buffer, the actual size of the buffer, in bytes, and where the buffer is located in memory.

TW_INFO

```
typedef struct {
      TW_UINT16        InfoID;
      TW_UINT16        ItemType;
      TW_UINT16        NumItems;
      TW_UINT16        CondCode;
      TW_UINT32        Item;
   } TW_INFO, FAR * pTW_ INFO;
```

Used by

Within TW_EXTIMAGEINFO structure.

Description

This structure is used to pass specific information between the data source and the application.

Field Descriptions

InfoID	Tag identifying an information. For TW_EXTIMAGEINFO, the information ID is defined as IACAP_xxxx. (Please refer to Extended Image capabilities).
ItemType	Item data type. It is one of TWTY_xxxx value.
NumItems	Number of items for this field.
CondCode	This is condition code of availability of data for extended image attribute requested. Following is the list of possible condition codes: TWRC_INFONOTSUPPORTED TWRC_DATANOTAVAILABLE
Item	Data Item. For anything that is <= 4 bytes, it is actual data. Otherwise it is a handle to data location. If < 4 bytes, it is 4 byte aligned.

Following is the list of added return codes.

TWRC_INFONOTSUPPORTED	Requested information is not supported.
TWRC_DATANOTAVAILABLE	Requested information is supported, but some unknown reason, information is not available.

TW_JPEGCOMPRESSION

```
typedef struct {
   TW_UINT16       ColorSpace;
   TW_UINT32       SubSampling;
   TW_UINT16       NumComponents;
   TW_UINT16       RestartFrequency;
   TW_UINT16       QuantMap[4]Manufacturer;
   TW_MEMORY       QuantTable[4];
   TW_UINT16       HuffmanMap[4];
   TW_MEMORY       HuffmanDC[2];
   TW_MEMORY       HuffmanAC[2];
} TW_JPEGCOMPRESSION, FAR * pTW_JPEGCOMPRESSION;
```

Used by

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GET
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GETDEFAULT
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_RESET
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_SET

Description

Describes the information necessary to transfer a JPEG-compressed image during a buffered transfer. Images compressed in this fashion will be compatible with the JPEG File Interchange Format, version 1.1. For more information on JPEG and TWAIN, see Chapter 4. The TWAIN JPEG implementation is based on the JPEG Draft International Standard, version 10918-1. The sample tables found in Section K of the JPEG Draft International Standard, version 10918-1 are used as the default tables for QuantTable, HuffmanDC, and HuffmanAC.

Field Descriptions

ColorSpace	One of the TWPT_xxxx values. Defines the color space in which the compressed components are stored. Only spaces supported by the Source for ICAP_JPEGPIXELTYPE are valid.
SubSampling	Encodes the horizontal and vertical subsampling in the form ABCDEFGH, where ABCD are the high-order four nibbles which represent the horizontal subsampling and EFGH are the low-order four nibbles which represent the vertical subsampling. Each nibble may have a value of 0, 1, 2, 3, or 4. However, max(A,B,C,D) * max(E,F,G,H) must be less than or equal to 10. Subsampling is irrelevant for single component images. Therefore, the corresponding nibbles should be set to 1. e.g. To indicate subsampling two Y for each U and V in a YUV space image, where the same subsampling occurs in both horizontal and vertical axes, this field would hold 0x21102110. For a grayscale image, this field would hold 0x10001000. A CMYK image could hold 0x11111111.
NumComponents	Number of color components in the image to be compressed.
RestartFrequency	Number of MDUs (Minimum Data Units) between restart markers. Default is 0, indicating that no restart markers are used. An MDU is defined for interleaved data (i.e. R-G-B, Y-U-V, etc.) as a minimum complete set of 8x8 component blocks.
QuantMap[4]	Mapping of components to Quantization tables.
QuantTable[4]	Quantization tables.
HuffmanMap[4]	Mapping of components to Huffman tables. Null entries signify selection of the default tables.
HuffmanDC[2]	DC Huffman tables. Null entries signify selection of the default tables.
HuffmanAC[2]	AC Huffman tables. Null entries signify selection of the default tables.

TW_MEMORY

```
typedef struct {
   TW_UINT32      Flags;
   TW_UINT32      Length;
   TW_MEMREF      TheMem;
} TW_MEMORY, FAR * pTW_MEMORY;
```

Used by

Embedded in the TW_IMAGEMEMXFER and TW_JPEGCOMPRESSION structures

Description

Provides information for managing memory buffers. Memory for transfer buffers is allocated by the application--the Source is asked to fill these buffers. This structure keeps straight which entity is responsible for deallocation.

Field Descriptions

Flags

Encodes which entity releases the buffer and how the buffer is referenced. The ownership flags must be used:

- when transferring Buffered Memory data as tiles
- when transferring Buffered Memory that is compressed
- in the TW_JPEGCOMPRESSION structure

When transferring Buffered Memory data as uncompressed strips, the application allocates the buffers and is responsible for setting the ownership flags.

This field is used to identify how the memory is to be referenced. The memory is always referenced by a Handle on the Macintosh and a Pointer under UNIX. It is referenced by a Handle or a pointer under Microsoft Windows.

Use TWMF_xxxx constants, bit-wise OR'd together to fill this field.

Flag Constants:

TWMF_APPOWNS	0x1
TWMF_DSMOWNS	0x2
TWMF_DSOWNS	0x4
TWMF_POINTER	0x8
TWMF_HANDLE	0x10

Length

The size of the buffer in bytes. Should always be an even number and word-aligned.

TheMem

Reference to the buffer. May be a Pointer or a Handle (see Flags field to make this determination). You must typecast this field before referencing it in your code.

TW_MEMREF

On Windows:

```
typedef LPVOID     TW_MEMREF;
```

On Macintosh:

```
typedef char       *TW_MEMREF;
```

On Unix:

```
typedef unsigned charEMREF;
```

Used by

Embedded in the TW_EVENT and TW_MEMORY structures

Description

Memory references are specific to each operating system. TWAIN defines TW_MEMREF to be the memory reference type supported by the operating system.

Field Descriptions

See definitions above

TW_ONEVALUE

```
typedef struct {
   TW_UINT16      ItemType;
   TW_UINT32      Item;
} TW_ONEVALUE, FAR * pTW_ONEVALUE;
```

Used by

TW_CAPABILITY structure (when ConType field specifies TWON_ONEVALUE)

Description

Stores a single value (item) which describes a capability. This structure is currently used only in a TW_CAPABILITY structure. Such a value would be useful to describe the current value of the device's contrast, or to set a specific contrast value. This structure is related in function and purpose to TW_ARRAY, TW_ENUMERATION, and TW_RANGE.

Note that in cases where the data type is TW_UINT16, the data should reside in the lower word.

Field Descriptions

ItemType	The type of the item. The type is indicated by the constant held in this field. The constant is of the kind TWTY_xxxx.
Item	The value.

TW_PALETTE8

```
typedef struct {
   TW_UINT16      NumColors;
   TW_UINT16      PaletteType;
   TW_ELEMENT8    Colors[256];
} TW_PALETTE8, FAR * pTW_PALETTE8;
```

Used by

DG_IMAGE / DAT_PALETTE8 / MSG_GET
DG_IMAGE / DAT_PALETTE8 / MSG_GETDEFAULT
DG_IMAGE / DAT_PALETTE8 / MSG_RESET
DG_IMAGE / DAT_PALETTE8 / MSG_SET

Description

This structure holds the color palette information for buffered memory transfers of type ICAP_PIXELTYPE = TWPT_PALETTE.

Field Descriptions

NumColors	Number of colors in the color table; maximum index into the color table should be one less than this (since color table indexes are zero-based).
PaletteType	TWPA_xxxx constant specifying the type of palette.
Colors[256]	Array of palette values.

TW_PASSTHRU

```
typedef struct {
    TW_MEMREF      pCommand;
    TW_UINT32      CommandBytes;
    TW_INT32       Direction;
    TW_MEMREF      pData;
    TW_UINT32      DataBytes;
    TW_UINT32      DataBytesXfered;
} TW_PASSTHRU, FAR * pTW_PASSTHRU;
```

Used by

DG_CONTROL / DAT_PASSTHRU / MSG_PASSTHRU

Description

Used to bypass the TWAIN protocol when communicating with a device. All memory must be allocated and freed by the Application. Use of this feature is limited to Source writers who require a standard entry point for specialized Applications, such as diagnostics.

Field Descriptions

pCommand	Pointer to Command buffer.
CommandBytes	Number of bytes in Command buffer.
Direction	One of the TWDR_xxxx values. Defines the direction of data flow.
pData	Pointer to Data buffer.
DataBytes	Number of bytes in Data buffer.
DataBytesXfered	Number of bytes successfully transferred.

TW_PENDINGXFERS

```
typedef struct {
   TW_UINT16 Count;
   union {
           TW_UINT32 EOJ;
           TW_UINT32 Reserved;
   };
} TW_PENDINGXFERS, FAR *pTW_PENDINGXFERS;
```

Used by

DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER
DG_CONTROL / DAT_PENDINGXFERS / MSG_GET
DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET

Description

This structure tells the application how many more complete transfers the Source currently has available. The application should MSG_GET this structure at the conclusion of a transfer to confirm the Source's current state. If the Source has more transfers pending it will remain in State 6 awaiting initiation of the next transfer by the application.

If it has no more image transfers pending, it will place zero into the Count and will have automatically transitioned to State 5 (audio transfers will remain in State 6, even when the Count goes to zero).

If the Source knows there are more transfers pending but is unsure of the actual number, it should place -1 into Count (for example, with document feeders or continuous video sources). Otherwise, the Source should place the actual number of pending transfers into Count.

Field Descriptions

Count	When DAT_XFERGROUP is set to DG_IMAGE The number of complete transfers a Source has available for the application it is connected to. If no more transfers are available, set to zero. If an unknown and non-zero number of transfers are available, set to -1. When DAT_XFERGROUP is set to DG_AUDIO The number of complete audio snippet transfers for a given image a Source has available for the application it is connected to. If no more transfers are available, set to zero. –1 is not a valid value.
EOJ	The application should check this field if the CAP_JOBCONTROL is set to other than TWJC_NULL. If the EOJ is not 0, the application should expect more data from the driver according to CAP_JOBCONTROL settings.
Reserved	Maintained so as not to cause compile time errors for pre-1.7 code.

TW_RANGE

```
typedef struct {
   TW_UINT16      ItemType;
   TW_UINT32      MinValue;
   TW_UINT32      MaxValue;
   TW_UINT32      StepSize;
   TW_UINT32      DefaultValue;
   TW_UINT32      CurrentValue;
} TW_RANGE, FAR * pTW_RANGE;
```

Used by

TW_CAPABILITY structure (when ConType field specifies TWON_RANGE)

Description

Stores a range of individual values describing a capability. The values are uniformly distributed between a minimum and a maximum value. The step size between each value is constant. Such a value is useful when describing such capabilities as the resolutions of a device which supports discreet, uniform steps between each value, such as 50 through 300 dots per inch in steps of 2 dots per inch (50, 52, 54, ..., 296, 298, 300). This structure is related in function and purpose to TW_ARRAY, TW_ENUMERATION, and TW_ONEVALUE.

Field Descriptions

ItemType	The type of items in the list. The type is indicated by the constant held in this field. The constant is of the kind TWTY_xxxx. All items in the list have the same size / type.
MinValue	The least positive / most negative value of the range.
MaxValue	The most positive / least negative value of the range.
StepSize	The delta between two adjacent values of the range. e.g. Item2 - Item1 = StepSize;
DefaultValue	The device’s “power-on” value for the capability. If the application is performing a MSG_SET operation and isn’t sure what the default value is, set this field to TWON_DONTCARE32.
CurrentValue	The value to which the device (or its user interface) is currently set to for the capability.

TW_RGBRESPONSE

```
typedef struct {
   ELEMENT8      Response[1];
} TW_RGBRESPONSE, FAR * pTW_RGBRESPONSE;
```

Used by

DG_IMAGE / DAT_RGBRESPONSE / MSG_RESET
DG_IMAGE / DAT_RGBRESPONSE / MSG_SET

Description

This structure is used by the application to specify a set of mapping values to be applied to RGB color data. Use this structure for RGB data whose bit depth is up to, and including, 8-bits. The number of elements in the array is determined by TW_IMAGEINFO.BitsPerPixel—the number of elements is 2 raised to the power of TW_IMAGEINFO.BitsPerPixel.

This structure is primarily intended for use by applications that bypass the Source's built-in user interface.

Field Descriptions

Response[1]	Transfer curve descriptors. To minimize color shift problems, writing the same values into each channel is desirable.

TW_SETUPFILEXFER

```
typedef struct {
   TW_STR255      FileName;
   TW_UINT16      Format;
   TW_INT16       VRefNum;
} TW_SETUPFILEXFER, FAR * pTW_SETUPFILEXFER;
```

Used by

DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET
DG_CONTROL / DAT_SETUPFILEXFER / MSG_GETDEFAULT
DG_CONTROL / DAT_SETUPFILEXFER / MSG_RESET
DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET

Description

Describes the file format and file specification information for a transfer through a disk file.

Field Descriptions

FileName	A complete file specifier to the target file. On Windows, be sure to include the complete pathname.
Format	The format of the file the Source is to fill. Fill with the correct constant—as negotiated with the Source—of type TWFF_xxxx.
VRefNum	The volume reference number for the file. This applies to Macintosh only. On Windows, fill the field with TWON_DONTCARE16.

TW_SETUPMEMXFER

```
typedef struct {
   TW_UINT32      MinBufSize;
   TW_UINT32      MaxBufSize;
   TW_UINT32      Preferred;
} TW_SETUPMEMXFER, FAR * pTW_SETUPMEMXFER;
```

Used by

DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET

Description

Provides the application information about the Source's requirements and preferences regarding allocation of transfer buffer(s). The best applications will allocate buffers of the Preferred size. An application should never allocate a buffer smaller than MinBufSize. Some Sources may not be able to fill a buffer larger than MaxBufSize so a larger allocation is a waste of RAM (digital cameras or frame grabbers fit this category).

Sources should fill out all three fields as accurately as possible. If a Source can fill an indeterminately large buffer (hand scanners might do this), put a -1 in MaxBufSize.

Field Descriptions

MinBufSize	The size of the smallest transfer buffer, in bytes, that a Source can be successful with. This will typically be the number of bytes in an uncompressed row in the block to be transferred. An application should never allocate a buffer smaller than this.
MaxBufSize	The size of the largest transfer buffer, in bytes, that a Source can fill. If a Source can fill an arbitrarily large buffer, it might set this field to negative 1 to indicate this (a hand-held scanner might do this, depending on how long its cord is). Other Sources, such as frame grabbers, cannot fill a buffer larger than a certain size. Allocation of a transfer buffer larger than this value is wasteful.
Preferred	The size of the optimum transfer buffer, in bytes. A smart application will allocate transfer buffers of this size, if possible. Buffers of this size will optimize the Source's performance. Sources should be careful to put reasonable values in this field. Buffers that are 10's of kbytes will be easier for applications to allocate than buffers that are 100's or 1000's of kbytes.

TW_STATUS

```
typedef struct {
   TW_UINT16      ConditionCode;
   TW_UINT16      Reserved;
} TW_STATUS, FAR * pTW_STATUS;
```

Used by

DG_CONTROL / DAT_STATUS / MSG_GET

Description

Used to describe the status of a Source. To ask the Source to fill in this structure, the application sends:

DG_CONTROL / DAT_STATUS / MSG_GET

with a pointer to a TW_STATUS structure. This is typically done in response to a Return Code other than TWRC_SUCCESS and should always be done in response to a Return Code of TWRC_CHECKSTATUS. In such a case, the Source has something it needs the application to know about.

Field Descriptions

ConditionCode	The TWCC_xxxx code (Condition Code) being returned to the application.
Reserved	Reserved for future use.

TW_TRANSFORMSTAGE

```
typedef struct {
   TW_DECODEFUNCTION      Decode[3];
   TW_FIX32               Mix[3][3];
} TW_TRANSFORMSTAGE, FAR * pTW_TRANSFORMSTAGE;
```

Used by

Embedded in the TW_CIECOLOR structure

Description

Specifies the parametrics used for either the ABC or LMN transform stages. This structure parallels the TTransformStage structure definition in Appendix A.

Field Descriptions

Decode[3]	Channel-specific transform parameters.
Mix[3][3]	3x3 matrix that specifies how channels are mixed in

TW_USERINTERFACE

```
typedef struct {
   TW_BOOL        ShowUI;
   TW_BOOL        ModalUI;
   TW_HANDLE      hParent;
} TW_USERINTERFACE, FAR * pTW_USERINTERFACE;
```

Used by

DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS
DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

Description

This structure is used to handle the user interface coordination between an application and a Source.

Field Descriptions

ShowUI	Set to TRUE by the application if the Source should activate its built-in user interface. Otherwise, set to FALSE. Note that not all sources support ShowUI = FALSE. See the description of DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS for more information.
ModalUI	Set to TRUE by the Source if the Source's built-in user interface behaves modally. This is described in Chapter 4 of this Toolkit.
hParent	Microsoft Windows only: Application's window handle. The Source designates the hWnd as its parent when creating the Source dialog. **NOTE:** Window handle allows Source's user interface to be a proper child of the parent application.

TW_VERSION

```
typedef struct {
   TW_UINT16      MajorNum;
   TW_UINT16      MinorNum;
   TW_UINT16      Language;
   TW_UINT16      Country;
   TW_STR32       Info;
} TW_VERSION, FAR * pTW_VERSION;
```

Used by

This is embedded in the TW_IDENTITY data structure

Description

A general way to describe the version of software that is running.

Field Descriptions

MajorNum	This refers to your application or Source's major revision number. e.g. The "2" in "version 2.01".
MinorNum	The incremental revision number of your application or Source. e.g. The "1" in "version 2.1".
Language	The primary language for your Source or application. e.g. TWLG_GER.
Country	The primary country where your Source or application is intended to be distributed. e.g. Germany.
Info	General information string - fill in as needed. e.g. "1.0b3 Beta release".

Extended Image Information Definitions

The following extended image attribute capabilities have been defined. If a data source wishes to create additional custom image attribute capabilities, it should define its TWEI_CUSTOM*xxx* identifiers with a base starting ID of TWEI_CUSTOM+(*x*) where *x* is a unique positive number defined by the data source.

For all extended image attributes see: DG_IMAGE / DAT_EXTIMAGEINFO / MSG_GET

Bar Code Recognition

TWEI_BARCODECOUNT

Description	Returns the number of bar codes found on the document image. A value of 0 means the bar code engine was enabled but that no bar codes were found. A value of -1 means the bar code engine was not enabled.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_BARCODECONFIDENCE

Description:	This number reflects the degree of certainty the bar code engine has in the accuracy of the information obtained from the scanned image and ranges from 0 (no confidence) to 100 (supreme confidence). The Source may return a value of -1 if it does not support confidence reporting.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_BARCODEROTATION

Description:	The bar code's orientation on the scanned image is described in reference to a Western-style interpretation of the image.	
Value Type:	TW_UINT32	
Allowed Values:	TWBCOR_ROT0	Normal reading orientation
	TWBCOR_ROT90	Rotated 90 degrees clockwise
	TWBCOR_ROT180	Rotated 180 degrees clockwise
	TWBCOR_ROT270	Rotated 270 degrees clockwise
	TWBCOR_ROTX	The orientation is not known.

TWEI_BARCODETEXTLENGTH

Description:	The number of ASCII characters derived from the bar code.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_BARCODETEXT

Description:	The text of a bar code found on a page.
Value Type:	TW_HANDLE
Allowed Values:	Any handle to a string

TWEI_BARCODEX

Description:	The X coordinate of a bar code found on a page.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_BARCODEY

Description:	The Y coordinate of a bar code found on a page.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _BARCODETYPE

Description:	The type of bar code found on a page.
Value Type:	TW_UINT32

Allowed Values:

TWBT_3OF9	0
TWBT_2OF5INTERLEAVED	1
TWBT_2OF5NONINTERLEAVED	2
TWBT_CODE93	3
TWBT_CODE128	4
TWBT_UCC128	5
TWBT_CODABAR	6
TWBT_UPCA	7
TWBT_UPCE	8
TWBT_EAN8	9
TWBT_EAN13	10
TWBT_POSTNET	11
TWBT_PDF417	12

Shaded Area Detection and Removal

TWEI _DESHADECOUNT

Description:	Returns the number of shaded regions found and erased in the document image. A value of 0 means the deshade engine was enabled but that no regions were processed. A value of -1 means the deshade engine was not enabled.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _DESHADETOP

Description:	The top coordinate of a shaded region found on a page.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _DESHADELEFT

Description:	The left coordinate of a shaded region found on a page.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _DESHADEHEIGHT

Description:	The height of a shaded region found on a page.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _DESHADEWIDTH

Description:	The width of a shaded region found on a page.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _DESHADESIZE

Description:	The width of the dots within the shade region.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_DESHADEBLACKCOUNTOLD

Description:	The total number of black pixels in the region prior to deshading. If this value is unknown the Source returns -1.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_ DESHADEBLACKCOUNTNEW

Description:	The total number of black pixels in the region after deshading. If this value is unknown the Source returns -1.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_ DESHADEBLACKRLMIN

Description:	The shortest black pixel run-length in the region prior to deshading. If this value is unknown the Source returns -1.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_ DESHADEBLACKRLMAX

Description:	The longest black pixel run-length in the region prior to deshading. If this value is unknown the Source returns -1.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_ DESHADEWHITECOUNTOLD

Description:	The total number of white pixels in the region prior to deshading. If this value is unknown the Source returns -1.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_ DESHADEWHITECOUNTNEW

Description:	The total number of white pixels in the region after deshading. If this value is unknown the Source returns -1.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_ DESHADEWHITERLMIN

Description:	The shortest white pixel run-length in the region prior to deshading. If this value is unknown the Source returns -1.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_ DESHADEWHITERLAVE

Description:	The average length of all white pixel run-lengths in the region prior to deshading. If this value is unknown the Source returns -1.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_ DESHADEWHITERLMAX

Description:	The longest white pixel run-length in the region prior to deshading. If this value is unknown the Source returns -1.
Value Type:	TW_UINT32
Allowed Values:	>=0

Speckle Removal

TWEI _SPECKLESREMOVED

Description:	The number of speckles removed from the image when de-speckle is enabled.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _BLACKSPECKLESREMOVED

Description:	The number of black speckles removed from the image when despeckle is enabled.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _WHITESPECKLESREMOVED

Description:	The number of white speckles removed (black speckles added) from the image when despeckle is enabled.
Value Type:	TW_UINT32
Allowed Values:	>=0

Horizontal Line Detection and Removal

TWEI _HORZLINECOUNT

Description:	Returns the number of horizontal lines found and erased in the document image. A value of 0 means the line removal engine was enabled but that no lines were found. A value of -1 means the line engine was not enabled.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _HORZLINEXCOORD

Description:	The x coordinate of a horizontal line detected in the image.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _HORZLINEYCOORD

Description:	The y coordinate of a horizontal line detected in the image.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _HORZLINELENGTH

Description:	The length of a horizontal line detected in the image.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _HORZLINETHICKNESS

Description:	The thickness (height) of a horizontal line detected in the image.
Value Type:	TW_UINT32
Allowed Values:	>=0

Vertical Line Detection and Removal

TWEI _VERTLINECOUNT

Description:	Returns the number of vertical lines found and erased in the document image. A value of 0 means the line removal engine was enabled but that no lines were found. A value of -1 means the line engine was not enabled.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _VERTLINEXCOORD

Description:	The x coordinate of a vertical line detected in the image.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _VERTLINEYCOORD

Description:	The y coordinate of a vertical line detected in the image.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI _VERTLINELENGTH

Description:	The length of a vertical line detected in the image.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_VERTLINETHICKNESS

Description:	The thickness (width) of a vertical line detected in the image.
Value Type:	TW_UINT32
Allowed Values:	>=0

Patch Code Detection (Job Separation)

TWEI_PATCHCODE

Description:	The patch code detected.	
Value Type:	TW_UINT32	
Allowed Values:	TWPCH_PATCH1	1
	TWPCH_PATCH2	2
	TWPCH_PATCH3	3
	TWPCH_PATCH4	4
	TWPCH_PATCH6	5
	TWPCH_PATCHT	6

Skew detection and Removal

TWEI_DESKEWSTATUS

Description:	Returns the status of the deskew operation.	
Value Type:	TW_UINT32	
Allowed Values:	TWDSK_SUCCESS	Image successfully deskewed
	TWDSK_REPORTONLY	Deskew information only
	TWDSK_FAIL	Deskew failed
	TWDSK_DISABLED	Deskew engine not enabled

TWEI_SKEWORIGINALANGLE

Description:	The amount of skew in the original image.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_SKEWFINALANGLE

Description:	The amount of skew in the deskewed image. This number may not be zero.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_SKEWCONFIDENCE

Description:	This number reflects the degree of certainty the deskew engine has in the accuracy of the deskewing of the current image and ranges from 0 (no confidence) to 100 (supreme confidence). The Source may return a value of -1 if it does not support confidence reporting.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_SKEWWINDOWX1

Description:	This is the X image coordinate of the upper left corner of the virtual deskewed image. It may be negative indicating the deskewed corner is not represented by actual pixels.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_SKEWWINDOWY1

Description:	The Y image coordinate of the upper left corner of the virtual deskewed image. It may be negative indicating the deskewed corner is not represented by actual pixels.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_SKEWWINDOWX2

Description:	The X image coordinate of the upper right corner of the virtual deskewed image.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_SKEWWINDOWY2

Description:	The Y image coordinate of the upper right corner of the virtual deskewed image.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_SKEWWINDOWX3

Description:	This is the X image coordinate of the lower left corner of the virtual deskewed image. It may be negative indicating the deskewed corner is not represented by actual pixels.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_SKEWWINDOWY3

Description:	The Y image coordinate of the lower left corner of the virtual deskewed image.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_SKEWWINDOWX4

Description:	The X image coordinate of the lower right corner of the virtual deskewed image.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_SKEWWINDOWY4

Description:	The Y image coordinate of the lower right corner of the deskewed image.
Value Type:	TW_UINT32
Allowed Values:	>=0

Endorsed / Imprinted Text

TWEI_ENDORSEDTEXT

Description:	The text that was endorsed on the paper by the scanner.
Value Type:	TW_STR255
Allowed Values:	Any string

Forms Recognition

TWEI_FORMCONFIDENCE

Description:	The confidence that the specified form was detected. This is an array property with a confidence factor for each form In the data set with 0 meaning no match and 100 meaning absolute certainty. Typically values over 70 imply a good form match with the template.
Value Type:	TW_UINT32
Allowed Values:	0 to 100

TWEI_FORMTEMPLATEMATCH

Description:	The array of file names for the master forms matched against a form. If multi-page master forms are used, the associated page numbers are contained in the FORMTEMPLATEPAGEMATCH capability array.
Value Type:	TW_STR255
Allowed Values:	Any string

TWEI_FORMTEMPLATEPAGEMATCH

Description:	An array containing the number of the page from a multi-page master form matched against a form image. It is useful when matching a form image against the pages of a multi-page master form. The file name of the master form is contained in the FORMTEMPLATEMATCH capability.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_FORMHORZDOCOFFSET

Description:	An array containing the perceived horizontal offsets of the form image being matched against a set of master forms. This is useful for page registration once the form has been recognized.
Value Type:	TW_UINT32
Allowed Values:	>=0

TWEI_FORMVERTDOCOFFSET

Description:	An array containing the perceived vertical offsets of the form image being matched against a set of master forms. This is useful for page registration once the form has been recognized.
Value Type:	TW_UINT32
Allowed Values:	>= 0

Data Argument Types that Don't Have Associated TW_Structures

Most of the DAT_xxxx components of the TWAIN operation triplets have a corresponding data structure whose name begins with TW_ and then uses the same suffix as the DAT_ name. However, the following do not use that pattern.

DAT_IMAGEFILEXFER

Acts on NULL data.

DAT_IMAGENATIVEXFER

Uses a TW_UINT32 variable.

- **On Windows:** In Win 3.1, the low word of this 32-bit integer is a handle variable to a DIB (Device Independent Bitmap) located in memory. For Win 95 the handles fill the entire field.
- **On Macintosh:** This 32-bit integer is a handle to a Picture (a PicHandle). It is a QuickDraw picture located in memory.

DAT_NULL

Used by the Source to signal the need for an event to announce MSG_XFERREADY or MSG_CLOSEDSREQ. (Used on Windows only)

DAT_PARENT

Used by the DG_CONTROL / DAT_PARENT / MSG_OPENDSM and MSG_CLOSEDSM operations.

- **On Windows:** They act on a variable of type TW_INT32. Prior to the operation, the application must write, a window handle to the application's window that acts as the "parent" for the Source's user interface. In Win 3.1 this would be in the low word, in Win 95 it will fill the entire field. (This must be done whether or not the Source's user interface will be used. The Source Manager uses this window handle to signal the application when data is ready for transfer (MSG_XFERREADY) or the Source needs to be closed (MSG_CLOSEDSREQ)).
- **On Macintosh:** These act on NULL data.

DAT_XFERGROUP

Used by the DG_CONTROL / DAT_XFERGROUP / MSG_GET operation. The data acted on by this operation is a variable of type TW_UINT32. (The same as a DG_xxxx designator.) The value of this variable is indeterminate prior to the operation. Following the operation, a single bit is set indicating the Data Group of the transfer.

Constants

Generic Constants

Constants	(defined as)	(values)
	TWON_PROTOCOLMINOR	0
	TWON_PROTOCOLMAJOR	1
	TWON_ARRAY	3
	TWON_ENUMERATION	4
	TWON_ONEVALUE	5
	TWON_RANGE	6
	TWON_ICONID	962
	TWON_DSMID	461
	TWON_DSMCODEID	63
	TWON_DONTCARE8	0xff
	TWON_DONTCARE16	0xffff
	TWON_DONTCARE32	0xffffffff
Flags used in TW_MEMORY		
	TWMF_APPOWNS	0x1
	TWMF_DSMOWNS	0x2
	TWMF_DSOWNS	0x4
	TWMF_POINTER	0x8
	TWMF_HANDLE	0x10
Palette types for TW_PALETTE8		
	TWPA_RGB	0
	TWPA_GRAY	1
	TWPA_CMY	2
Events for TW_DEVICEEVENT		
	TWDE_AUTOMATICCAPTURE	0
	TWDE_CHECKBATTERY	1
	TWDE_CHECKFLASH	2
	TWDE_CHECKPOWERSUPPLY	3
	TWDE_CHECKRESOLUTION	4
	TWDE_DEVICEADDED	5
	TWDE_DEVICEOFFLINE	6
	TWDE_DEVICEREADY	7
	TWDE_DEVICEREMOVED	8
	TWDE_PAPERDOUBLEFEED	9
	TWDE_PAPERJAM	10

File Types for TW_FILESYSTEM

TWFT_CAMERA	0
TWFT_CAMERATOP	1
TWFT_CAMERABOTTOM	2
TWFT_CAMERAPREVIEW	3
TWFT_DOMAIN	4
TWFT_HOST	5
TWFT_DIRECTORY	6
TWFT_IMAGE	7
TWFT_UNKNOWN	8

ItemTypes for Capability Container structures

TWTY_INT8	0x0000
TWTY_INT16	0x0001
TWTY_INT32	0x0002
TWTY_UINT8	0x0003
TWTY_UINT16	0x0004
TWTY_UINT32	0x0005
TWTY_BOOL	0x0006
TWTY_FIX32	0x0007
TWTY_FRAME	0x0008
TWTY_STR32	0x0009
TWTY_STR64	0x000a
TWTY_STR128	0x000b
TWTY_STR255	0x000c

Capability Constants

CAP_CLEARBUFFERS		
	TWCB_AUTO	0
	TWCB_CLEAR	1
	TWCB_NOCLEAR	2
CAP_POWERSUPPLY		
	TWPS_EXTERNAL	0
	TWPS_BATTERY	1
ICAP_BITDEPTHREDUCTION values		**(defined as)**
	TWBR_THRESHOLD	0
	TWBR_HALFTONES	1
	TWBR_CUSTHALFTONE	2
	TWBR_DIFFUSION	3
ICAP_BITORDER values		
	TWBO_LSBFIRST	0
	TWBO_MSBFIRST	1
ICAP_COMPRESSION values		
	TWCP_NONE	0
	TWCP_PACKBITS	1
	TWCP_GROUP31D	2
	TWCP_GROUP31DEOL	3
	TWCP_GROUP32D	4
	TWCP_GROUP34	5
	TWCP_JPEG	6
	TWCP_LZW	7
	TWCP_JBIG	8
	TWCP_PNG	9
	TWCP_RLE4	10
	TWCP_RLE8	11
	TWCP_BITFIELDS	12
ICAP_FILTER values		
	TWFT_RED	0
	TWFT_GREEN	1
	TWFT_BLUE	2
	TWFT_NONE	3
	TWFT_WHITE	4
	TWFT_CYAN	5
	TWFT_MAGENTA	6
	TWFT_YELLOW	7
	TWFT_BLACK	8
ICAP_FLASHUSED2		
	TWFL_NONE	0
	TWFL_OFF	1
	TWFL_ON	2
	TWFL_AUTO	3
	TWFL_REDEYE	4

ICAP_IMAGEFILEFORMAT values

TWFF_TIFF	0
TWFF_PICT	1
TWFF_BMP	2
TWFF_XBM	3
TWFF_JFIF	4
TWFF_FPX	5
TWFF_TIFFMULTI	6
TWFF_PNG	7
TWFF_SPIFF	8
TWFF_EXIF	9

ICAP_IMAGEFILTER

TWIF_NONE	0
TWIF_AUTO	1
TWIF_LOWPASS	2
TWIF_BANDPASS	3
TWIF_HIGHPASS	4

ICAP_LIGHTPATH values

TWLP_REFLECTIVE	0
TWLP_TRANSMISSIVE	1

ICAP_LIGHTSOURCE values

TWLS_RED	0
TWLS_GREEN	1
TWLS_BLUE	2
TWLS_NONE	3
TWLS_WHITE	4
TWLS_UV	5
TWLS_IR	6

ICAP_NOISEFILTER

TWNF_NONE	0
TWNF_AUTO	1
TWNF_LONEPIXEL	2
TWNF_MAJORITYRULE	3

ICAP_ORIENTATION values

TWOR_ROT0	0
TWOR_ROT90	1
TWOR_ROT180	2
TWOR_ROT270	3
TWOR_PORTRAIT	TWOR_ROT0
TWOR_LANDSCAPE	TWOR_ROT270

ICAP_OVERSCAN

TWOV_NONE	0
TWOV_AUTO	1
TWOV_TOPBOTTOM	2
TWOV_LEFTRIGHT	3
TWOV_ALL	4

ICAP_PLANARCHUNKY values

TWPC_CHUNKY	0
TWPC_PLANAR	1

ICAP_PIXELFLAVOR values

TWPF_CHOCOLATE	0
TWPF_VANILLA	1

ICAP_PIXELTYPE values

TWPT_BW	0
TWPT_GRAY	1
TWPT_RGB	2
TWPT_PALETTE	3
TWPT_CMY	4
TWPT_CMYK	5
TWPT_YUV	6
TWPT_YUVK	7
TWPT_CIEXYZ	8
TWPT_LAB	9

ICAP_SUPPORTEDSIZES values

TWSS_NONE	0
TWSS_A4LETTER	1
TWSS_B5LETTER	2
TWSS_USLETTER	3
TWSS_USLEGAL	4
TWSS_A5	5
TWSS_B4	6
TWSS_B6	7
/ / removed	8
TWSS_USLEDGER	9
TWSS_USEXECUTIVE	10
TWSS_A3	11
TWSS_B3	12
TWSS_A6	13
TWSS_C4	14
TWSS_C5	15
TWSS_C6	16
/ / 1.8 Additions	
TWSS_4A0	17
TWSS_2A0	18
TWSS_A0	19
TWSS_A1	20
TWSS_A2	21
TWSS_A4	TWSS_A4LETTER
TWSS_A7	22
TWSS_A8	23
TWSS_A9	24
TWSS_A10	25
TWSS_ISOB0	26
TWSS_ISOB1	27
TWSS_ISOB2	28
TWSS_ISOB3	TWSS_B3
TWSS_ISOB4	TWSS_B4
TWSS_ISOB5	29
TWSS_ISOB6	TWSS_B6
TWSS_ISOB7	30
TWSS_ISOB8	31
TWSS_ISOB9	32
TWSS_ISOB10	33
TWSS_JISB0	34

TWSS_JISB1	35
TWSS_JISB2	36
TWSS_JISB3	37
TWSS_JISB4	38
TWSS_JISB5	TWSS_B5LETTER
TWSS_JISB6	39
TWSS_JISB7	40
TWSS_JISB8	41
TWSS_JISB9	42
TWSS_JISB10	43
TWSS_C0	44
TWSS_C1	45
TWSS_C2	46
TWSS_C3	47
TWSS_C7	48
TWSS_C8	49
TWSS_C9	50
TWSS_C10	51
TWSS_USEXECUTIVE	52
TWSS_BUSINESSCARD	53

ICAP_XFERMECH values

TWSX_NATIVE	0
TWSX_FILE	1
TWSX_MEMORY	2

ICAP_UNITS values

TWUN_INCHES	0
TWUN_CENTIMETERS	1
TWUN_PICAS	2
TWUN_POINTS	3
TWUN_TWIPS	4
TWUN_PIXELS	5

Language Constants

Language

	(defined as)
TWLG_USERLOCALE	-1
TWLG_DAN	0
TWLG_DUT	1
TWLG_ENG	2
TWLG_FCF	3
TWLG_FIN	4
TWLG_FRN	5
TWLG_GER	6
TWLG_ICE	7
TWLG_ITN	8
TWLG_NOR	9
TWLG_POR	10
TWLG_SPA	11
TWLG_SWE	12
TWLG_USA	13
TWLG_AFRIKAANS	14

TWLG_ALBANIA	15
TWLG_ARABIC	16
TWLG_ARABIC_ALGERIA	17
TWLG_ARABIC_BAHRAIN	18
TWLG_ARABIC_EGYPT	19
TWLG_ARABIC_IRAQ	20
TWLG_ARABIC_JORDAN	21
TWLG_ARABIC_KUWAIT	22
TWLG_ARABIC_LEBANON	23
TWLG_ARABIC_LIBYA	24
TWLG_ARABIC_MOROCCO	25
TWLG_ARABIC_OMAN	26
TWLG_ARABIC_QATAR	27
TWLG_ARABIC_SAUDIARABIA	28
TWLG_ARABIC_SYRIA	29
TWLG_ARABIC_TUNISIA	30
TWLG_ARABIC_UAE	31
TWLG_ARABIC_YEMEN	32
TWLG_BASQUE	33
TWLG_BYELORUSSIAN	34
TWLG_BULGARIAN	35
TWLG_CATALAN	36
TWLG_CHINESE	37
TWLG_CHINESE_HONGKONG	38
TWLG_CHINESE_PRC	39
TWLG_CHINESE_SINGAPORE	40
TWLG_CHINESE_SIMPLIFIED	41
TWLG_CHINESE_TAIWAN	42
TWLG_CHINESE_TRADITIONAL	43
TWLG_CROATIA	44
TWLG_CZECH	45
TWLG_DANISH	TWLG_DAN
TWLG_DUTCH	TWLG_DUT
TWLG_DUTCH_BELGIAN	46
TWLG_ENGLISH	TWLG_ENG
TWLG_ENGLISH_AUSTRALIAN	47
TWLG_ENGLISH_CANADIAN	48
TWLG_ENGLISH_IRELAND	49
TWLG_ENGLISH_NEWZEALAND	50
TWLG_ENGLISH_SOUTHAFRICA	51
TWLG_ENGLISH_UK	52
TWLG_ENGLISH_USA	TWLG_USA
TWLG_ESTONIAN	53
TWLG_FAEROESE	54
TWLG_FARSI	55
TWLG_FINNISH	TWLG_FIN
TWLG_FRENCH	TWLG_FRN
TWLG_FRENCH_BELGIAN	56
TWLG_FRENCH_CANADIAN	TWLG_FCF
TWLG_FRENCH_LUXEMBOURG	57
TWLG_FRENCH_SWISS	58
TWLG_GERMAN	TWLG_GER
TWLG_GERMAN_AUSTRIAN	59
TWLG_GERMAN_LUXEMBOURG	60
TWLG_GERMAN_LIECHTENSTEIN	61
TWLG_GERMAN_SWISS	62
TWLG_GREEK	63

TWLG_HEBREW	64
TWLG_HUNGARIAN	65
TWLG_ICELANDIC	TWLG_ICE
TWLG_INDONESIAN	66
TWLG_ITALIAN	TWLG_ITN
TWLG_ITALIAN_SWISS	67
TWLG_JAPANESE	68
TWLG_KOREAN	69
TWLG_KOREAN_JOHAB	70
TWLG_LATVIAN	71
TWLG_LITHUANIAN	72
TWLG_NORWEGIAN	TWLG_NOR
TWLG_NORWEGIAN_BOKMAL	73
TWLG_NORWEGIAN_NYNORSK	74
TWLG_POLISH	75
TWLG_PORTUGUESE	TWLG_POR
TWLG_PORTUGUESE_BRAZIL	76
TWLG_ROMANIAN	77
TWLG_RUSSIAN	78
TWLG_SERBIAN_LATIN	79
TWLG_SLOVAK	80
TWLG_SLOVENIAN	81
TWLG_SPANISH	TWLG_SPA
TWLG_SPANISH_MEXICAN	82
TWLG_SPANISH_MODERN	83
TWLG_SWEDISH	TWLG_SWE
TWLG_THAI	84
TWLG_TURKISH	85
TWLG_UKRANIAN	86
TWLG_ASSAMESE	87
TWLG_BENGALI	88
TWLG_BIHARI	89
TWLG_BODO	90
TWLG_DOGRI	91
TWLG_GUJARATI	92
TWLG_HARYANVI	93
TWLG_HINDI	94
TWLG_KANNADA	95
TWLG_KASHMIRI	96
TWLG_MALAYALAM	97
TWLG_MARATHI	98
TWLG_MARWARI	99
TWLG_MEGHALAYAN	100
TWLG_MIZO	101
TWLG_NAGA	102
TWLG_ORISSI	103
TWLG_PUNJABI	104
TWLG_PUSHTU	105
TWLG_SERBIAN_CYRILLIC	106
TWLG_SIKKIMI	107
TWLG_SWEDISH_FINLAND	108
TWLG_TAMIL	109
TWLG_TELUGU	110
TWLG_TRIPURI	111
TWLG_URDU	112
TWLG_VIETNAMESE	113

9

Capabilities

Chapter Contents

Overview

Sources may support a large number of capabilities but are required to support very few. To determine if a capability is supported by a Source, the application can query the Source using a DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, or MSG_GETDEFAULT operation. The application specifies the particular capability by storing its identifier in the Cap field of the TW_CAPABILITY structure. This is the structure pointed to by the pData parameter in the DSM_Entry( ) call.

DG_CONTROL / DAT_CAPABILITY operations for capability negotiation include:

MSG_GET	Returns the available settings for this capability, as well as the Current and Default settings (if the container is TW_ENUMERATION or TW_RANGE).
MSG_GETCURRENT	Returns the Current setting for this capability.
MSG_GETDEFAULT	Returns the value of the Source's preferred Default values.
MSG_RESET	Returns the capability to its TWAIN Default (power-on) condition (i.e. all previous negotiation is ignored).
MSG_SET	Allows the application to set the Current value of a capability or even to restrict the available values to some subset of the Source's power-on set of values. Sources are strongly encouraged to allow the application to set as many of its capabilities as possible, and further to reflect these changes in the Source's user interface. This will ensure that the user can only select images with characteristics that are useful to the consuming application.

Required Capabilities

The list of required capabilities can be found in Chapter 5.

Sources must implement and make available to TWAIN applications the advertised features of the devices they support. This is especially true in “no-UI mode.” Thus, when a capability is listed as required by none, a Source must still support it if its device supports it.

Capabilities in Categories of Functionality

Asynchronous Device Events

CAP_DEVICEEVENT	MSG_SET selects which events the application wants the source to report; MSG_RESET returns the preferred settings of the source.

Audible Alarms

CAP_ALARMS	Turns specific audible alarms on and off.
CAP_ALARMVOLUME	Controls the volume of a device's audible alarm.

Audio

ACAP_AUDIOFILEFORMAT	Informs application which audio file formats the source can generate.
ACAP_XFERMECH	Allows application and source to identify which audio transfer mechanisms they have in common.

Automatic Adjustments

ICAP_AUTOMATICBORDERDETECTION	Turns automatic border detection on and off.
ICAP_AUTOMATICDESKEW	Turns automatic skew correction on and off.
ICAP_AUTODISCARDBLANKPAGES	
ICAP_AUTOMATICROTATE	When TRUE, depends on source to automatically rotate the image.
ICAP_FLIPROTATION	Orients images that flip orientation every other image.

Automatic Capture

CAP_AUTOMATICCAPTURE	Specifies the number of images to automatically capture.
CAP_TIMEBEFOREFIRSTCAPTURE	Selects the number of seconds before the first picture taken.
CAP_TIMEBETWEENCAPTURES	Selects the hundredths of a second to wait between pictures taken.

Automatic Scanning

CAP_AUTOSCAN	Enables the source's automatic document scanning process.
CAP_CLEARBUFFERS	MSG_GET reports presence of data in scanner's buffers; MSG_SET clears the buffers.
CAP_MAXBATCHBUFFERS	Describes the number of pages that the scanner can buffer when CAP_AUTOSCAN is enabled.

Bar Code Detection Search Parameters

ICAP_BARCODEDETECTIONENABLED	Turns bar code detection on and off.
ICAP_SUPPORTEDBARCODETYPES	Provides a list of bar code types that can be detected by current data source.
ICAP_BARCODEMAXSEARCHPRIORITIES	Specifies the maximum number of supported search priorities.
ICAP_BARCODESEARCHPRIORITIES	A prioritized list of bar code types dictating the order in which they will be sought.
ICAP_BARCODESEARCHMODE	Restricts bar code searching to certain orientations, or prioritizes one orientation over another.
ICAP_BARCODEMAXRETRIES	Restricts the number of times a search will be retried if no bar codes are found.
ICAP_BARCODETIMEOUT	Restricts the total time spent on searching for bar codes on a page.

Capability Negotiation Parameters

CAP_EXTENDEDCAPS	Capabilities negotiated in States 5 & 6
CAP_SUPPORTEDCAPS	Inquire Source’s capabilities valid for MSG_GET

Color

ICAP_FILTER	Color characteristics of the subtractive filter applied to the image data
ICAP_GAMMA	Gamma correction value for the image data
ICAP_PLANARCHUNKY	Color data format - Planar or Chunky

Compression

ICAP_BITORDERCODES	CCITT Compression
ICAP_CCITTKFACTOR	CCITT Compression
ICAP_COMPRESSION	Compression method for Buffered Memory Transfers
ICAP_JPEGPIXELTYPE	JPEG Compression
ICAP_PIXELFLAVORCODES	CCITT Compression
ICAP_TIMEFILL	CCITT Compression

Device Parameters

CAP_DEVICEONLINE	Determines if hardware is on and ready
CAP_DEVICETIMEDATE	Date and time of a device's clock.
CAP_SERIALNUMBER	The serial number of the currently selected source device.
ICAP_EXPOSURETIME	Exposure time used to capture the image, in seconds
ICAP_FLASHUSED2	For devices that support a flash, MSG_SET selects the flash to be used; MSG_GET reports the current setting.
ICAP_IMAGEFILTER	For devices that support image filtering, selects the algorithm to be used.
ICAP_LAMPSTATE	Is the lamp on?
ICAP_LIGHTPATH	Image was captured transmissively or reflectively
ICAP_LIGHTSOURCE	Describes the color characteristic of the light source used to acquire the image
ICAP_NOISEFILTER	For devices that support noise filtering, selects the algorithm to be used.
ICAP_OVERSCAN	For devices that support overscanning, controls whether additional rows or columns are appended to the image.
ICAP_PHYSICALHEIGHT	Maximum height Source can acquire (in ICAP_UNITS)
ICAP_PHYSICALWIDTH	Maximum width Source can acquire (in ICAP_UNITS)
ICAP_UNITS	Unit of measure (inches, centimeters, etc.)
ICAP_ZOOMFACTOR	With MSG_GET, returns all camera supported lens zooming range.

Imprinter/Endorser Functionality

CAP_PRINTER	MSG_GET returns current list of available printer devices; MSG_SET selects the device for negotiation.
CAP_PRINTERENABLED	Turns the current CAP_PRINTER device on or off.
CAP_PRINTERINDEX	Starting number for the CAP_PRINTER device.
CAP_PRINTERMODE	Specifies appropriate current CAP_PRINTER device mode.
CAP_PRINTERSTRING	String(s) to be used in the string component when CAP_PRINTER device is enabled.
CAP_PRINTERSUFFIX	String to be used as current CAP_PRINTER device's suffix.

Image Information

CAP_AUTHOR	Author of acquired image (may include a copyright string)
CAP_CAPTION	General note about acquired image
CAP_TIMEDATE	Date and Time the image was acquired (entered State 7)

Image Parameters for Acquire

ICAP_AUTOBRIGHT	Enable Source's Auto-brightness function
ICAP_BRIGHTNESS	Source brightness values
ICAP_CONTRAST	Source contrast values
ICAP_HIGHLIGHT	Lightest highlight, values lighter than this value will be set to this value
ICAP_ORIENTATION	Defines which edge of the paper is the top: Portrait or Landscape
ICAP_ROTATION	Source can, or should, rotate image this number of degrees
ICAP_SHADOW	Darkest shadow, values darker than this value will be set to this value
ICAP_XSCALING	Source Scaling value (1.0 = 100%) for x-axis
ICAP_YSCALING	Source Scaling value (1.0 = 100%) for y-axis

Image Type

ICAP_BITDEPTH	Pixel bit depth for Current value of ICAP_PIXELTYPE
ICAP_BITDEPTHREDUCTION	Allows a choice of the reduction method for bit depth loss
ICAP_BITORDER	Specifies how the bytes in an image are filled by the Source
ICAP_CUSTHALFTONE	Square-cell halftone (dithering) matrix to be used
ICAP_HALFTONES	Source halftone patterns
ICAP_PIXELFLAVOR	Sense of the pixel whose numeric value is zero
ICAP_PIXELTYPE	The type of pixel data (B/W, gray, color, etc.)
ICAP_THRESHOLD	Specifies the dividing line between black and white values

Language Support

CAP_LANGUAGE	Allows application and source to identify which languages they have in common.

Pages

ICAP_FRAMES	Size and location of frames on page
ICAP_MAXFRAMES	Maximum number of frames possible per page
ICAP_SUPPORTEDSIZES	Fixed frame sizes for typical page sizes

Paper Handling

CAP_AUTOFEED	MSG_SET to TRUE to enable Source's automatic feeding
CAP_CLEARPAGE	MSG_SET to TRUE to eject current page and leave acquire area empty
CAP_FEEDERALIGNMENT	If TRUE, feeder is centered; FALSE if it is free-floating.
CAP_FEEDERENABLED	If TRUE, Source's feeder is available
CAP_FEEDERLOADED	If TRUE, Source has documents loaded in feeder (MSG_GET only)
CAP_FEEDERORDER	Specifies whether feeder starts with top of first or last page.
CAP_FEEDPAGE	MSG_SET to TRUE to eject current page and feed next page
CAP_PAPERBINDING	
CAP_PAPERDETECTABLE	Determines whether source can detect documents on the ADF or flatbed.
CAP_REACQUIREALLOWED	Indicates whether the physical hardware is capable of acquiring multiple images of the same page without changes to the physical registration of that page.
CAP_REWINDPAGE	MSG_SET to TRUE to do a reverse feed

Patch Code Detection

ICAP_PATCHCODEDETECTIONENABLED	Turns patch code detection on and off.
ICAP_SUPPORTEDPATCHCODETYPES	List of patch code types that can be detected by current data source.
ICAP_PATCHCODEMAXSEARCHPRIORITIES	Maximum number of search priorities.
ICAP_PATCHCODESEARCHPRIORITIES	List of patch code types dictating the order in which patch codes will be sought.
ICAP_PATCHCODESEARCHMODE	Restricts patch code searching to certain orientations, or prioritizes one orientation over another.
ICAP_PATCHCODEMAXRETRIES	Restricts the number of times a search will be retried if none are found on a page.
ICAP_PATCHCODETIMEOUT	Restricts total time for searching for a patch code on a page.

Power Monitoring

CAP_BATTERYMINUTES	The minutes of battery power remaining on a device.
CAP_BATTERYPERCENTAGE	With MSG_GET, indicates battery power status.
CAP_POWERSAVETIME	With MSG_SET, sets the camera power down timer in seconds; with MSG_GET, returns the current setting of the power down time.
CAP_POWERSUPPLY	MSG_GET reports the kinds of power available; MSG_GETCURRENT reports the current power supply to use.

Resolution

ICAP_XNATIVERESOLUTION	Native optical resolution of device for x-axis
ICAP_XRESOLUTION	Current / Available optical resolutions for x-axis
ICAP_YNATIVERESOLUTION	Native optical resolution of device for y-axis
ICAP_YRESOLUTION	Current / Available optical resolutions for y-axis

Transfers

CAP_XFERCOUNT	Number of images the application is willing to accept this session
ICAP_COMPRESSION	Buffered Memory transfer compression schemes
ICAP_IMAGEFILEFORMAT	File formats for file transfers
ICAP_TILES	Tiled image data
ICAP_XFERMECH	Transfer mechanism - used to learn options and set-up for upcoming transfer
ICAP_UNDEFINEDIMAGESIZE	The application will accept undefined image size

User Interface

CAP_CAMERAPREVIEWUI	Queries the source for UI support for preview mode.
CAP_ENABLEDSUIONLY	Queries an application to see if it implements the new user interface settings dialog.
CAP_INDICATORS	Use the Source's progress indicator? (valid only when ShowUI==FALSE)
CAP_UICONTROLLABLE	Indicates that Source supports acquisitions with UI disabled

The Capability Listings

The following section lists descriptions of all TWAIN capabilities in alphabetical order. The format of each capability entry is:

NAME OF CAPABILITY

Description

Description of the capability

Application

(Optional) Information for the application

Source

(Optional) Information for the Source

Values

Type:	Data structure for the capability.
Default Value:	The value the Source must use as the Current value when entering State 4 (following DG_CONTROL / DAT_IDENTITY / MSG_OPENDS). This is the value the Source resets the Current value to when it receives a MSG_RESET operation. The Source reports its preferred Default value when it receives a MSG_GETDEFAULT. The Source's preferred value may be different from the TWAIN Default value.
Allowed Values:	Definition of the values allowed for this capability.
Container for MSG_GET	Acceptable containers for use on MSG_GET operations.
Container for MSG_SET	Acceptable containers for use on MSG_SET operations.

Required By

If a Source or application is required to support the capability.

Source Required Operations

Operations the Source is required to support.

See Also

Associated capabilities and data structures.

ACAP_AUDIOFILEFORMAT

Description

Informs the application which audio file formats the Source can generate (MSG_GET). Tells the Source which audio file formats the application can handle (MSG_SET).

Application

Use this ACAP to determine which formats are available for audio file transfers, but use the DG_CONTROL / DAT_SETUPAUDIOFILEXFER / MSG_SET operation to specify the format to be used for a particular acquisition.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	TWAF_WAV (Windows)	
	TWAF_AIFF (Macintosh)	
Allowed Values:	TWAF_WAV	0
	TWAF_AIFF	1
	TWAF_AU	3
	TWAF_SND	4
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION	
	TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

DG_CONTROL / DAT_SETUPAUDIOFILEXFER / MSG_SET
DG_AUDIO / DAT_AUDIOFILEXFER / MSG_GET

ACAP_XFERMECH

Description

Allows the Application and Source to identify which audio transfer mechanisms they have in common.

Application

The current setting of ACAP_XFERMECH must match the constant used by the application to specify the audio transfer mechanism when starting the transfer using the triplet: DG_AUDIO / DAT_AUDIOxxxxXFER / MSG_GET.

Values

Type:	TW_UINT16	
Default Value:	TWSX_NATIVE	
Allowed Values:	TWSX_NATIVE	0
	TWSX_FILE	1
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION	
	TW_ONEVALUE	

Required By

All Audio Sources

Source Required Operations

MSG_GET / CURRENT / DEFAULT,
MSG_SET / RESET

See Also

DG_AUDIO / DAT_AUDIOxxxxXFER / MSG_GET

CAP_ALARMS

Description

Turns specific audible alarms on and off.

Application

Note that an application may opt to turn off all alarms by issuing a MSG_SET with no data. Therefore, an application should also be prepared to receive an empty array from a Source with an MSG_GET. (i.e., pTW_ARRAY->NumItems == 0)

The easiest way to test for allowed values is to try to set them all with MSG_SET. If not all are allowed, the Source will return TWCC_CHECKSTATUS with those values that it supports.

Source

It is worth noting that the alarms do not have to be present in the device for a Source to make use of this capability. If the device is capable of alerting the Source to these various kinds of conditions, but is unable to generate the alarms, itself; then the Source may opt to generate them on its behalf.

TWAL_ALARM is a catchall for alarms not explicitly listed. It is also used where a device only provides control over a single, multi-use alarm. For instance, if a device beeps for both jams and bar-codes, but doesn't allow independent control of the alarms, then it should report TWAL_ALARM to cover them, and not TWAL_BARCODE, TWAL_JAM.

TWAL_FEEDERERROR covers paper handling errors such as jams, double-feeds, skewing and the like; conditions that most likely stop scanning.

TWAL_FEEDERWARNING covers non-fatal events, such as feeder empty.

TWAL_DOUBLEFEED, TWAL_JAM and TWALSKEW cover paper handling errors.

TWAL_BARCODE and TWAL_PATCHCODE generate alarms when an image with this kind of data is recognized.

TWAL_POWER generates alarms for any changes in power to the device.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	No default	
Allowed Values:	TWAL_ALARM	0
	TWAL_FEEDERERROR	1
	TWAL_FEEDERWARNING	2
	TWAL_BARCODE	3
	TWAL_DOUBLEFEED	4
	TWAL_JAM	5
	TWAL_PATCHCODE	6
	TWAL_POWER	7
	TWAL_SKEW	8
Container for MSG_GET:	TW_ARRAY	
Container for MSG_SET:	TW_ARRAY	

Required By

None

Source Required Operations

None

See Also

CAP_ALARMVOLUME

CAP_ALARMVOLUME

Description

The volume of a device's audible alarm. Note that this control affects the volume of all alarms; no specific volume control for individual types of alarms is provided.

Application

Take note of the range step, some Sources may only offer a step of 100, which turns the alarm on or off.

Source

If 0, the audible alarm is turned off. All other values control the volume of the alarm.

Windows only - If the alarm is managed in the Source, as opposed to the device, then it should be consistent with the control panel Accessibility Options (i.e., the user should get visual notification if that is the current setting for the desktop).

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_INT32
Default Value:	No default
Allowed Values:	0 - 100
Container for MSG_GET:	TW_ONEVALUE, TW_RANGE
Container for MSG_SET:	TW_ONEVALUE, TW_RANGE

Required By

None

Source Required Operations

None

See Also

CAP_ALARMS

CAP_AUTHOR

Description

The name or other identifying information about the Author of the image. It may include a copyright string.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_STR128
Default Value:	“\0”
Allowed Values:	Any string
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_CAPTION
CAP_TIMEDATE

CAP_AUTOFEED

Description

If TRUE, the Source will automatically feed the next page from the document feeder after the number of frames negotiated for capture from each page are acquired. CAP_FEEDERENABLED must be TRUE to use this capability.

Application

Set the capability to TRUE to enable the Source's automatic feed process, or FALSE to disable it. After the completion of each transfer, check TW_PENDINGXFERS. Count to determine if the Source has more images to transfer. A -1 means there are more images to transfer but the exact number is not known.

CAP_FEEDERLOADED indicates whether the Source's feeder is loaded. (The automatic feed process continues whenever this capability is TRUE.)

Source

If CAP_FEEDERENABLED equals FALSE, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED (capability is not supported in current settings).

If it is supported, return TWRC_SUCCESS and enable the device's automatic feed process: After all frames negotiated for capture from each page are acquired, put the current document in the output area and advance the next document from the input area to the feeder image acquisition area. If the feeder input area is empty, the automatic feeding process is suspended but should continue when the feeder is reloaded.

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

All Sources with Feeder Devices

Source Required Operations

MSG_GET / CURRENT / DEFAULT,
MSG_SET / RESET

See Also

CAP_CLEARPAGE
CAP_FEEDERENABLED
CAP_FEEDERLOADED
CAP_FEEDPAGE
CAP_REWINDPAGE

CAP_AUTOMATICCAPTURE

Description

The number of images to automatically capture. This does *not* refer to the number of images to be sent to the Application, use CAP_XFERCOUNT for that.

Source

If 0, Automatic Capture is disabled If 1 or greater, that number of images is captured by the device.

Automatic capture implies that the device is capable of capturing images without the presence of the Application. This means that it must be possible for the Application to close the Source and reopen it later, after the images have been captured.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_INT32
Default Value:	0
Allowed Values:	0 or greater
Container for MSG_GET:	TW_ONEVALUE, TW_RANGE
Container for MSG_SET:	TW_ONEVALUE, TW_RANGE

Required By

None

Source Required Operations

None

See Also

CAP_TIMEBEFOREFIRSTCAPTURE
CAP_TIMEBETWEENCAPTURES
CAP_XFERCOUNT

DG_CONTROL / DAT_FILESYSTEM / MSG_AUTOMATICCAPTUREDIRECTORY

CAP_AUTOSCAN

Description

This capability is intended to boost the performance of a Source. The fundamental assumption behind AutoScan is that the device is able to capture the number of images indicated by the value of CAP_XFERCOUNT without waiting for the Application to request the image transfers. This is only possible if the device has internal buffers capable of caching the images it captures.

The default behavior is undefined, because some high volume devices are incapable of anything but CAP_AUTOSCAN being equal to TRUE. However, if a Source supports FALSE, it should use it as the mandatory default, since this best describes the behavior of pre-1.8 TWAIN Applications.

Application

The application should check the TW_PENDINGXFERS.Count, and continue to scan until it becomes 0.

When AutoScan is set to TRUE, the Application should not rely on just the paper sensors (for example, CAP_FEEDERLOADED) to determine if there are images to be transferred. The latency between the Source and the Application makes it very likely that at the time the sensor reports FALSE, there may be more than one image waiting for the transfer inside of the device's buffers. Applications should use the TW_PENDINGXFERS.Count returned from DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER to determine whether or not there are more images to be transferred.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL /DAT_CAPABILITY/ MSG_QUERYSUPPORT)

Values

Type:	TW_BOOL
Default Value:	No default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_AUTOFEED
CAP_CLEARBUFFERS
CAP_MAXBATCHBUFFERS

CAP_BATTERYMINUTES

Description

The minutes of battery power remaining to the device.

Source

-2 indicates that the available power is infinite.

-1 indicates that the device cannot report the remaining battery power.

0 and greater indicates the minutes of battery life remaining.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_INT32
Default Value:	No default
Allowed Values:	-2, -1, 0, and greater
Container for MSG_GET:	TW_ONEVALUE, TW_RANGE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

None

See Also

CAP_POWERSUPPLY,
CAP_BATTERYPERCENTAGE

CAP_BATTERYPERCENTAGE

Description

When used with MSG_GET, return the percentage of battery power level on camera. If -1 is returned, it indicates that the battery is not present.

Application

Use this capability with MSG_GET to indicate to the user about the battery power status. It is recommended to use CAP_POWERSUPPLY to identify the power source first.

Source

-2 indicates that the available power is infinite.

-1 indicates that the device cannot report the remaining battery power.

0 to 100 indicates the percentage of battery life remaining.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_INT16
Default Value:	None
Allowed Values:	-2, -1, 0 to 100.
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	Not allowed

Required By

None. Highly recommended for digital cameras that are equipped with batteries.

Source Required Operations

MSG_GET

See Also

CAP_POWERSUPPLY,
CAP_BATTERYMINUTES

CAP_CAMERAPREVIEWUI

Description

This capability queries the Source for UI support for preview mode. If TRUE, the Source supports preview UI.

Application

Use this capability to query the preview UI support by the Source. However, the application can choose to use the Source's UI or not even if the Source supports it.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_BOOL
Default Value:	None
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	Not allowed.

Required By

None. Highly recommended for digital cameras.

Source Required Operations

MSG_GET

CAP_CAPTION

Description

A general note about the acquired image.

Source

If not supported, the Source should return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_STR255
Default Value:	"\0"
Allowed Values:	Any string
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_AUTHOR
CAP_TIMEDATE

CAP_CLEARBUFFERS

Description

MSG_GET reports the presence of data in the scanner's buffers. MSG_SET with a value of TWCB_CLEAR immediately clears the buffers.

Source

MSG_SET: TWCB_AUTO causes the Source to automatically clear the buffers when it transitions from state 4 to state 5, or from state 5 to state 4.

MSG_SET: TWCB_CLEAR causes the Source to immediately clear its buffers.

MSG_SET: TWCB_NOCLEAR causes the Source to preserve images in the buffers. If the Source transitions from state 4 to state 5 with images in its buffer, it will immediately report MSG_XFERREADY, and deliver those images before any new images scanned by the user.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	TWCB_AUTO	
Allowed Values:	TWCB_AUTO	0
	TWCB_CLEAR	1
	TWCB_NOCLEAR	2
Container for MSG_GET:	TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

CAP_AUTOSCAN
CAP_MAXBATCHBUFFERS

CAP_CLEARPAGE

Description

If TRUE, the Source will eject the current page being acquired from and will leave the feeder acquire area empty.

If CAP_AUTOFEED is TRUE, a fresh page will be advanced.

CAP_FEEDERENABLED must equal TRUE to use this capability.

This capability must have been negotiated as an extended capability to be used in States 5 and 6.

Application

Do a MSG_SET on this capability to advance the document in the feeder acquire area to the output area and abort all transfers pending on this page.

This capability is used in States 5 and 6 by applications controlling the Source's feeder (usually without the Source user interface).

This capability can also be used while CAP_AUTOFEED equals TRUE to abort all remaining transfers on this page and continue with the next page.

Source

If CAP_FEEDERENABLED equals FALSE, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED (capability is not supported in current settings).

If supported, advance the document in the feeder-acquire area to the output area and abort all pending transfers from this page.

The Source will perform this action once whenever the capability is MSG_SET to TRUE. The Source should then revert the Current value to FALSE.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_AUTOFEED
CAP_EXTENDEDCAPS
CAP_FEEDERENABLED
CAP_FEEDERLOADED
CAP_FEEDPAGE
CAP_REWINDPAGE

CAP_CUSTOMDSDATA

Description

Allows the application to query the data source to see if it supports the new operation triplets DG_CONTROL / DAT_CUSTOMDSDATA / MSG_GET and DG_CONTROL / DAT_CUSTOMDSDATA / MSG_SET.

If TRUE, the source will support the DG_CONTROL / DAT_CUSTOMDSDATA / MSG_GET message.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	Set not allowed

Required By

None

Source Required Operations

None

See Also

DG_CONTROL / DAT_CUSTOMDSDATA / MSG_GET

CAP_DEVICEEVENT

Description

MSG_SET selects which events the Application wants the Source to report. MSG_GET gets the current setting. MSG_RESET resets the capability to the empty array (no events set).

TWDE_CHECKAUTOMATICCAPTURE:	The automatic capture settings on the device have been changed by the user.
TWDE_CHECKBATTERY:	The status of the battery has changed.
TWDE_CHECKFLASH:	The flash setting on the device has been changed by the user.
TWDE_CHECKPOWERSUPPLY:	The power supply has been changed (for instance, the user may have just connected AC to a device that was running on battery power).
TWDE_CHECKRESOLUTION:	The x/y resolution setting on the device has been changed by the user.
TWDE_DEVICEADDED:	The user has added a device (for instance a memory card in a digital camera).
TWDE_DEVICEOFFLINE:	A device has become unavailable, but has not been removed.
TWDE_DEVICEREADY:	The device is ready to capture an image.
TWDE_DEVICEREMOVED:	The user has removed a device.
TWDE_IMAGECAPTURED:	The user has captured an image to the device's internal storage.
TWDE_IMAGEDELETED:	The user has removed an image from the device's internal storage.
TWDE_PAPERDOUBLEFEED:	Two or more sheets of paper have been fed together.
TWDE_PAPERJAM:	The device's document feeder has jammed.
TWDE_LAMPFAILURE:	The device's light source has failed.
TWDE_CHECKDEVICEONLINE:	The device has been turned off and on.
TWDE_POWERSAVE:	The device has powered down to save energy.
TWDE_POWERSAVENOTIFY:	The device is about to power down to save energy.
TWDE_CUSTOMEVENTS:	Baseline for events specific to a given Source.

Application

Set all values and process the TWRC_FAILURE / TWCC_CHECKSTATUS (if returned) to identify those items supported by the Source.

Source

The startup default must be an empty array. Generate TWRC_FAILURE / TWCC_CHECKSTATUS and remove unsupported events when an Application requests events not supported by the Source.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Please note that the actions of an Application must never directly generate a device event. For instance, if the user deletes an image using the controls on the device, then the Source should generate an event. If, however, an Application deletes an image in the device (using DG_CONTROL / DAT_FILESYSTEM / MSG_DELETE), then the Source must not generate an event.

Values

Type:	TW_UINT16	
Default Value:	(empty array)	
Allowed Values:	TWDE_CHECKAUTOMATICCAPTURE	0
	TWDE_CHECKBATTERY	1
	TWDE_CHECKFLASH	2
	TWDE_CHECKPOWERSUPPLY	3
	TWDE_CHECKRESOLUTION	4
	TWDE_DEVICEADDED	5
	TWDE_DEVICEOFFLINE	6
	TWDE_DEVICEREADY	7
	TWDE_DEVICEREMOVED	8
	TWDE_IMAGECAPTURED	9
	TWDE_IMAGEDELETED	10
	TWDE_PAPERDOUBLEFEED	11
	TWDE_PAPERJAM	12
	TWDE_LAMPFAILURE	13
	TWDE_POWERDOWNNOTIFY	14
	TWDE_CUSTOMEVENTS	0x8000
Container for MSG_GET:	TW_ARRAY	
Container for MSG_SET:	TW_ARRAY	

Required By

None

Source Required Operations

None

See Also

DG_CONTROL / DAT_NULL / MSG_DEVICEEVENT
DG_CONTROL / DAT_DEVICEEVENT / MSG_GET

Device Events Article

CAP_DEVICEONLINE

Description

If TRUE, the physical hardware (e.g., scanner, digital camera, image database, etc.) that represents the image source is attached, powered on, and communicating.

Application

This capability can be issued at any time to determine the availability of the image source hardware.

Source

The receipt of this capability request should trigger a test of the status of the physical link to the image source. The source should not assume that the link is still active since the last transaction, but should issue a transaction that actively tests this condition.

Values

Type:	TW_BOOL
Default Value:	None
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

All image Sources

Source Required Operations

MSG_GET / CURRENT / DEFAULT

CAP_DEVICETIMEDATE

Description

The date and time of the device's clock.

Managed in the form "YYYY/MM/DD HH:mm:SS:sss" where YYYY is the year, MM is the numerical month, DD is the numerical day, HH is the hour, mm is the minute, SS is the second, and sss is the millisecond.

Source

The internal date and time of the device. Be sure to leave the space between the ending of the date and the beginning of the time fields. All fields must be specified for MSG_SET.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL /DAT_CAPABILITY/ MSG_QUERYSUPPORT)

Values

Type:	TW_STR32
Default Value:	No Default
Allowed Values:	Any date
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_TIMEDATE

CAP_DUPLEX

Description

This indicates whether the scanner supports duplex. If so, it further indicates whether one-path or two-path duplex is supported.

Application

Application can send MSG_GET to find out whether the scanner supports duplex.

Source

Source should determine level of duplex support returning the values accordingly.

Values

Type:	TW_UINT16
Default Value:	TWDX_NONE
Allowed Values:	TWDX_NONE TWDX_1PASSDUPLEX TWDX_2PASSDUPLEX
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	Not allowed.

Required By

None

Source Required Operations

See Also

CAP_DUPLEXENABLED

CAP_DUPLEXENABLED

Description

The user can set the duplex option to be TRUE or FALSE. If TRUE, the scanner scans both sides of a paper; otherwise, the scanner will scan only one side of the image.

Application

Application should send MSG_GET to determine if the duplex option is enabled or not.

Source

Source should return TRUE or FALSE based on the level of duplex support; otherwise, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

See Also

CAP_DUPLEX

CAP_ENABLEDSUIONLY

Description

Allows an application to query a source to see if it implements the new user interface settings dialog. If a source reports that it has the capability CAP_ENABLEDSUIONLY, then it must implement the operation triplet DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDSUIONLY to display the source user interface without acquiring an image.

If TRUE, the source will support the DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDSUIONLY message.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	Set not allowed

Required By

None.

Source Required Operations

None

See Also

DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDSUIONLY

CAP_ENDORSER

Description

Allows the application to specify the starting endorser / imprinter number. All other endorser/ imprinter properties should be handled through the data source's user interface.

The user can set the starting number for the endorser.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT32
Default Value:	0
Allowed Values:	Any value
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

See Also

None

CAP_EXTENDEDCAPS

Description

Allows the application and Source to negotiate capabilities to be used in States 5 and 6.

Application

MSG_GETCURRENT provides a list of all capabilities which the Source and application have agreed to negotiate in States 5 and 6.

MSG_GET provides a list of all capabilities the Source is willing to negotiate in States 5 and 6.

MSG_SET specifies which capabilities the application wants to negotiate in States 5 and 6.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16
Default Value:	No Default
Allowed Values:	Any *x*CAP_*xxxx*
Container for MSG_GET:	TW_ARRAY
Container for MSG_SET:	TW_ARRAY

Required By

None

Source Required Operations

None

See Also

CAP_SUPPORTEDCAPS

CAP_FEEDERALIGNMENT

Description

Helps the Application determine any special actions it may need to take when negotiating frames with the Source.

TWFA_NONE:	The alignment is free-floating. Applications should assume that the origin for frames is on the left.
TWFA_LEFT:	The alignment is to the left.
TWFA_CENTER:	The alignment is centered. This means that the paper will be fed in the middle of the ICAP_PHYSICALWIDTH of the device. If this is set, then the Application should calculate any frames with a left offset of zero.
TWFA_RIGHT:	The alignment is to the right.

Application

The Application can use this to determine if it must center the framing information sent to the Source. With some Sources it might be possible for the Application to select whether the paper is center fed or not.

Source

Use this capability to report the state of the feeder.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWFA_NONE	0
	TWFA_LEFT	1
	TWFA_CENTER	2
	TWFA_RIGHT	3
Container for MSG_GET:	TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE, if supported	

Required By

None

Source Required Operations

None

CAP_FEEDERENABLED

Description

If TRUE, Source must acquire data from the document feeder acquire area and other feeder capabilities can be used. If FALSE, Source must acquire data from the non-feeder acquire area and no other feeder capabilities can be used.

Application

The application should MSG_SET this capability to TRUE before attempting to use any other feeder capabilities. This sets the current acquire area to the feeder area (it may not be a different physical area on some Sources).

The application can MSG_SET this capability to FALSE to use the Source's non-feeder acquisition area and disallow the further use of feeder capabilities.

Source

This setting should reflect the current acquire area:

If TRUE, feeder acquire area should be used

If FALSE, use non-feeder acquire area

Usually, the feeder acquire area and non-feeder acquire area of the Source will be the same. For example, a flatbed scanner may feed a page onto the flatbed platen then scanning always takes place from the platen.

The counter example is a flatbed scanner that moves the scan bar over the platen when CAP_FEEDERENABLED is FALSE, but moves the paper over the scan bar when it is TRUE.

Default Support Guidelines for Sources

- Flatbed scanner (without an optional ADF installed) - Default to FALSE. Do not allow setting to TRUE (return TWRC_FAILURE / TWCC_BADVALUE) but support the capability (never return TWRC_FAILURE / TWCC_CAPUNSUPPORTED).
- A device that uses the same acquire area for feeder and non-feeder, and has a feeder installed - Default to TRUE and allow settings to TRUE or FALSE (meaning allow or don't allow other feeder capabilities).
- A device that operates differently when acquiring from the feeder and non-feeder areas (for example, physical pages sizes are different) - Default to preferred area and allow setting to either TRUE or FALSE.
- A sheet feed scanner or image database - Default to TRUE (meaning there is only one acquire area - the feeder area) and do not allow setting to FALSE (return TWRC_FAILURE / TWCC_BADVALUE).
- A handheld scanner would not support this capability (return TWRC_FAILURE / TWCC_CAPUNSUPPORTED).

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

All Sources with feeder devices

Source Required Operations

MSG_GET / CURRENT / DEFAULT

See Also

CAP_AUTOFEED
CAP_CLEARPAGE
CAP_FEEDERLOADED
CAP_FEEDPAGE
CAP_REWINDPAGE

CAP_FEEDERLOADED

Description

Reflect whether there are documents loaded in the Source's feeder.

Application

Used by application to inquire whether there are documents loaded in the Source's feeder.

CAP_FEEDERENABLED must equal TRUE to use this capability.

Source

If CAP_FEEDERENABLED equals FALSE, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED (capability is not supported in current settings).

If CAP_FEEDERENABLED equals TRUE, return the status of the feeder (documents loaded = TRUE; no documents loaded = FALSE).

The Source is responsible for reporting instructions to users on using the device. This includes instructing the user to place documents in the feeder when CAP_FEEDERLOADED equals FALSE and the application has requested a feed page (manually or automatically).

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

All Sources with feeder devices

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

CAP_AUTOFEED
CAP_CLEARPAGE
CAP_FEEDERENABLED
CAP_FEEDPAGE
CAP_REWINDPAGE

CAP_FEEDERORDER

Description

TWFO_FIRSTPAGEFIRST if the feeder starts with the top of the first page.
TWFO_LASTPAGEFIRST is the feeder starts with the top of the last page.

Application

An Application can use this to determine if it should reorganize the stream of images received from a Source.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWFO_FIRSTPAGEFIRST	0
	TWFO_LASTPAGEFIRST	1
Container for MSG_GET:	TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE, if supported	

Required By

None

Source Required Operations

None

See Also

CAP_FEEDERENABLED

CAP_FEEDPAGE

Description

If TRUE, the Source will eject the current page and advance the next page in the document feeder into the feeder acquire area.

If CAP_AUTOFEED is TRUE, the same action just described will occur and CAP_AUTOFEED will remain active.

CAP_FEEDERENABLED must equal TRUE to use this capability.

This capability must have been negotiated as an extended capability to be used in States 5 and 6.

Application

Do a MSG_SET to TRUE on this capability to advance the next document in the feeder to the feeder acquire area.

This capability is used in States 5 and 6 by applications controlling the Source's feeder (usually without the Source's user interface).

This capability can also be used while CAP_AUTOFEED equals TRUE to abort all remaining transfers on this page and continue with the next page.

Source

If CAP_FEEDERENABLED equals FALSE, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED (capability is not supported in current settings).

If supported, advance the document in the feeder-acquire area to the output area and abort all pending transfers from this page.

Advance the next page in the input area to the feeder acquire area. If there are no documents in the input area, return: TWRC_FAILURE / TWCC_BADVALUE.

The Source will perform this action once whenever the capability is MSG_SET to TRUE. The Source should then revert the Current value to FALSE.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_AUTOFEED
CAP_CLEARPAGE
CAP_EXTENDEDCAPS
CAP_FEEDERENABLED
CAP_FEEDERLOADED
CAP_REWINDPAGE

CAP_INDICATORS

Description

If TRUE, the Source will display a progress indicator during acquisition and transfer, regardless of whether the Source's user interface is active. If FALSE, the progress indicator will be suppressed if the Source's user interface is inactive.

The Source will continue to display device-specific instructions and error messages even with the Source user interface and progress indicators turned off.

Application

If the application plans to enable the Source with TW_USERINTERFACE. ShowUI = FALSE, it can also suppress the Source's progress indicator by using this capability.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_BOOL
Default Value:	TRUE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

CAP_JOBCONTROL

Description

Allows multiple jobs in batch mode. The application can decide how the job can be processed, according to the flags listed below.

TWJC_NONE	No job control.
TWJC_JSIC	Detect and include job separator and continue scanning.
TWJC_JSIS	Detect and include job separator and stop scanning.
TWJC_JSXC	Detect and exclude job separator and continue scanning.
TWJC_JSXS	Detect and exclude job separator and stop scanning.

If application selects options other than none, it should check the JCL field of the new PENDINGXFERS data.

To distinguish between jobs, a job separator sheet containing patch code can be inserted. If the application knows the how to save different jobs, the TWJC_JSIC or TWJC_JSXC can be used. When this job separator is detected, the application will give a separate name for each job. If the application does not know how to save different jobs, it can use TWJC_JSIS or TWJC_JSXS to stop scanning and ask the user for different job name.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16
Default Value:	TWJC_NONE
Allowed Values:	TWJC_NONE TWJC_JSIC TWJC_JSIS TWJC_JSXC TWJC_JSXS
Container for MSG_GET:	TW_ONEVALUE / TW_ENUMERATION
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

MSG_PENDINGXFER

CAP_LANGUAGE

Description

Allows Application and Source to identify which languages they have in common for the exchange of string data, and to select the language of the internal UI.

Note: Since the TWLG_xxxx codes include language and country data, there is no separate capability for selecting the country.

Application

In multi-lingual environments, it is the responsibility of the Application to recall the last selected language for a given User.

Source

The current value of this setting specifies the language used by the Source (and possibly the device). The Source must first default to the Application's current language. If that fails then it must default to the User's Locale (c.f., the Win32 call GetLocaleInfo()). If that fails then the Source should make the best choice it can, preferably using a common secondary language (i.e., English, French…).

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Note:

- TWLG_ARABIC_UAE is for the United Arabic Emirates.
- TWLG_CHINESE_PRC is for the People's Republic of China

Values

Type:	TW_UINT16	
Default Value:	**In order of priority:** 1) appIdentity->Version.Language 2) TWLG_USERLOCALE 3) Source's choice	
Allowed Values:	TWLG_USERLOCALE	-1
	// pre 1.8 values…	
	TWLG_DAN	0
	TWLG_DUT	1
	TWLG_ENG	2
	TWLG_FCF	3
	TWLG_FIN	4
	TWLG_FRN	5
	TWLG_GER	6
	TWLG_ICE	7

TWLG_ITN	8
TWLG_NOR	9
TWLG_POR	10
TWLG_SPA	11
TWLG_SWE	12
TWLG_USA	13
// 1.8 should use these…	
TWLG_AFRIKAANS	14
TWLG_ALBANIA	15
TWLG_ARABIC	16
TWLG_ARABIC_ALGERIA	17
TWLG_ARABIC_BAHRAIN	18
TWLG_ARABIC_EGYPT	19
TWLG_ARABIC_IRAQ	20
TWLG_ARABIC_JORDAN	21
TWLG_ARABIC_KUWAIT	22
TWLG_ARABIC_LEBANON	23
TWLG_ARABIC_LIBYA	24
TWLG_ARABIC_MOROCCO	25
TWLG_ARABIC_OMAN	26
TWLG_ARABIC_QATAR	27
TWLG_ARABIC_SAUDIARABIA	28
TWLG_ARABIC_SYRIA	29
TWLG_ARABIC_TUNISIA	30
TWLG_ARABIC_UAE	31
TWLG_ARABIC_YEMEN	32
TWLG_BASQUE	33
TWLG_BYELORUSSIAN	34
TWLG_BULGARIAN	35
TWLG_CATALAN	36
TWLG_CHINESE	37
TWLG_CHINESE_HONGKONG	38
TWLG_CHINESE_PRC	39
TWLG_CHINESE_SINGAPORE	40
TWLG_CHINESE_SIMPLIFIED	41
TWLG_CHINESE_TAIWAN	42
TWLG_CHINESE_TRADITIONAL	43
TWLG_CROATIA	44
TWLG_CZECH	45
TWLG_DANISH	TWLG_DAN
TWLG_DUTCH	TWLG_DUT
TWLG_DUTCH_BELGIAN	46
TWLG_ENGLISH	TWLG_ENG
TWLG_ENGLISH_AUSTRALIAN	47
TWLG_ENGLISH_CANADIAN	48
TWLG_ENGLISH_IRELAND	49
TWLG_ENGLISH_NEWZEALAND	50
TWLG_ENGLISH_SOUTHAFRICA	51
TWLG_ENGLISH_UK	52
TWLG_ENGLISH_USA	TWLG_USA
TWLG_ESTONIAN	53

TWLG_FAEROESE	54
TWLG_FARSI	55
TWLG_FINNISH	TWLG_FIN
TWLG_FRENCH	TWLG_FRN
TWLG_FRENCH_BELGIAN	56
TWLG_FRENCH_CANADIAN	TWLG_FCF
TWLG_FRENCH_LUXEMBOURG	57
TWLG_FRENCH_SWISS	58
TWLG_GERMAN	TWLG_GER
TWLG_GERMAN_AUSTRIAN	59
TWLG_GERMAN_LUXEMBOURG	60
TWLG_GERMAN_LIECHTENSTEIN	61
TWLG_GERMAN_SWISS	62
TWLG_GREEK	63
TWLG_HEBREW	64
TWLG_HUNGARIAN	65
TWLG_ICELANDIC	TWLG_ICE
TWLG_INDONESIAN	66
TWLG_ITALIAN	TWLG_ITN
TWLG_ITALIAN_SWISS	67
TWLG_JAPANESE	68
TWLG_KOREAN	69
TWLG_KOREAN_JOHAB	70
TWLG_LATVIAN	71
TWLG_LITHUANIAN	72
TWLG_NORWEGIAN	TWLG_NOR
TWLG_NORWEGIAN_BOKMAL	73
TWLG_NORWEGIAN_NYNORSK	74
TWLG_POLISH	75
TWLG_PORTUGUESE	TWLG_POR
TWLG_PORTUGUESE_BRAZIL	76
TWLG_ROMANIAN	77
TWLG_RUSSIAN	78
TWLG_SERBIAN_LATIN	79
TWLG_SLOVAK	80
TWLG_SLOVENIAN	81
TWLG_SPANISH	TWLG_SPA
TWLG_SPANISH_MEXICAN	82
TWLG_SPANISH_MODERN	83
TWLG_SWEDISH	TWLG_SWE
TWLG_THAI	84
TWLG_TURKISH	85
TWLG_UKRANIAN	86
TWLG_ASSAMESE	87
TWLG_BENGALI	88
TWLG_BIHARI	89
TWLG_BODO	90
TWLG_DOGRI	91
TWLG_GUJARATI	92
TWLG_HARYANVI	93
TWLG_HINDI	94

TWLG_KANNADA	95
TWLG_KASHMIRI	96
TWLG_MALAYALAM	97
TWLG_MARATHI	98
TWLG_MARWARI	99
TWLG_MEGHALAYAN	100
TWLG_MIZO	101
TWLG_NAGA	102
TWLG_ORISSI	103
TWLG_PUNJABI	104
TWLG_PUSHTU	105
TWLG_SERBIAN_CYRILLIC	106
TWLG_SIKKIMI	107
TWLG_SWEDISH_FINLAND	108
TWLG_TAMIL	109
TWLG_TELUGU	110
TWLG_TRIPURI	111
TWLG_URDU	112
TWLG_VIETNAMESE	113

Container for MSG_GET: TW_ENUMERATION, TW_ONEVALUE

Container for MSG_SET: TW_ENUMERATION, TW_ONEVALUE

Required By

None

Source Required Operations

None

CAP_MAXBATCHBUFFERS

Description

Describes the number of pages that the scanner can buffer when CAP_AUTOSCAN is enabled.

Application

MSG_GET returns the supported values

MSG_SET sets the current number pages to be buffered (if the Source allows this to be set)

Source

If supported, report the maximum batch buffer settings during MSG_GET. If MSG_SET is supported, limit batch buffers to the requested value for future transfers.

If not supported, return TWRC_FAILURE/ TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT32
Default Value:	No Default
Allowed Values:	1 to 2^{32} –1
Container for MSG_GET:	TW_ONEVALUE TW_ENUMERATION TW_RANGE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_AUTOSCAN
CAP_CLEARBUFFERS

CAP_REACQUIREALLOWED

Description

Indicates whether the physical hardware (e.g. scanner, digital camera) is capable of acquiring multiple images of the same page without changes to the physical registration of that page.

Application

Use this capability to enable or disable modes of operation where multiple image acquisitions of the page are required. Examples: preview mode, automated image analysis mode.

Source

If supported, return TRUE if the device is capable of capturing the page image multiple times without refeeding the page or otherwise causing physical registration changes. Return FALSE otherwise.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Support Guidelines for Sources

- A flat bed scanner that can retain the page on the platen and moves the scan bar past the page would return TRUE.
- A sheet-fed scanner that physically moves the page past the scan bar would return FALSE.
- A hand held scanner would return FALSE.

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

None

See Also

CAP_AUTOFEED
CAP_CLEARPAGE
CAP_FEEDERENABLED
CAP_FEEDPAGE
CAP_REWINDPAGE

CAP_PAPERDETECTABLE

Description

This capability determines whether the device has a paper sensor that can detect documents on the ADF or Flatbed.

Application

If the source returns FALSE, the application should not rely on values such as CAP_FEEDERLOADED, and continue as if the paper is loaded.

Source

If supported, the source is responsible for detecting whether document is loaded or not.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_BOOL
Default Value:	TRUE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

None

See Also

CAP_FEEDERLOADED

CAP_POWERSAVETIME

Description

When used with MSG_SET, set the camera power down timer in seconds. When used with MSG_GET, return the current setting of the power down time.

Application

Use this capability with MSG_SET to set the user selected camera power down time, when no activity is detected by the camera. The default value of -1 means no power down, power is always on.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_INT32
Default Value:	-1
Allowed Values:	>= -1
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None. Highly recommended for digital cameras. MSG_GET, MSG_SET, MSG_RESET

Source Required Operations

CAP_POWERSUPPLY

Description

MSG_GET reports the kinds of power available to the device. MSG_GETCURRENT reports the current power supply in use.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16
Default Value:	No default
Allowed Values:	TWPS_EXTERNAL TWPS_BATTERY
Container for MSG_GET:	TW_ENUMERATION, TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

None

CAP_PRINTER

Description

MSG_GET returns the current list of available printer devices, along with the one currently being used for negotiation. MSG_SET selects the current device for negotiation, and optionally constrains the list. MSG_RESET restores all the available devices (useful after MSG_SET has been used to constrain the list).

Top/Bottom refer to duplex devices, and indicate if the printer is writing on the top or the bottom of the sheet of paper. Simplex devices use the top settings.

Before/After indicates whether printing occurs before or after the sheet of paper has been scanned.

Application

Use this capability to determine which printers are available for negotiation, and to select a specific printer prior to negotiation.

Source

Imprinters are used to print data on documents at the time of scanning, and may be used for any purpose. Endorsers are more specific in nature, stamping some kind of proof of scanning on the document. Applications may opt to use imprinters for endorsing documents.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWPR_IMPRINTERTOPBEFORE	0
	TWPR_IMPRINTERTOPAFTER	1
	TWPR_IMPRINTERBOTTOMBEFORE	2
	TWPR_IMPRINTERBOTTOMAFTER	3
	TWPR_ENDORSERTOPBEFORE	4
	TWPR_ENDORSERTOPAFTER	5
	TWPR_ENDORSERBOTTOMBEFORE	6
	TWPR_ENDORSERBOTTOMAFTER	7
Container for MSG_GET:	TW_ENUMERATION, TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION, TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

CAP_PRINTERENABLED
CAP_PRINTERINDEX
CAP_PRINTERMODE
CAP_PRINTERSTRING
CAP_PRINTERSUFFIX

CAP_PRINTERENABLED

Description

Turns the current CAP_PRINTER device on or off.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_PRINTER
CAP_PRINTERINDEX
CAP_PRINTERMODE
CAP_PRINTERSTRING
CAP_PRINTERSUFFIX

CAP_PRINTERINDEX

Description

The User can set the starting number for the current CAP_PRINTER device.

Source

This value allows the user to set the starting page number for the current CAP_PRINTER device.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT32
Default Value:	0
Allowed Values:	Any values.
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

See Also

CAP_PRINTER
CAP_PRINTERENABLED
CAP_PRINTERMODE
CAP_PRINTERSTRING
CAP_PRINTERSUFFIX

CAP_PRINTERMODE

Description

Specifies the appropriate current CAP_PRINTER device mode.

Note:

- TWPM_SINGLESTRING specifies that the printed text will consist of a single string.
- TWPM _MULTISTRING specifies that the printed text will consist of an enumerated list of strings to be printed in order.
- TWPM _COMPOUNDSTRING specifies that the printed string will consist of a compound of a String followed by a value followed by a suffix string.

Application

Negotiate this capability to specify the mode of printing to use when the current CAP_PRINTER device is enabled.

Source

If supported, use the specified mode for future image acquisitions.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16
Default Value:	TWPM_SINGLESTRING
Allowed Values:	TWPM_SINGLESTRING TWPM_MULTISTRING TWPM_COMPOUNDSTRING
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_PRINTER
CAP_PRINTERENABLED
CAP_PRINTERINDEX
CAP_PRINTERSTRING
CAP_PRINTERSUFFIX

CAP_PRINTERSTRING

Description

Specifies the string(s) that are to be used in the string component when the current CAP_PRINTER device is enabled.

Application

Negotiate this capability to specify the string or strings to be used for printing (depending on printer mode). Use enumeration to print multiple lines of text, one line per string in the enumerated list. Be sure to check the status codes if attempting multiple lines, since not all devices support this feature.

Source

If supported, use the specified string for printing during future acquisitions.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_STR255
Default Value:	No Default
Allowed Values:	Any string
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_PRINTER
CAP_PRINTERENABLED
CAP_PRINTERINDEX
CAP_PRINTERMODE
CAP_PRINTERSUFFIX

CAP_PRINTERSUFFIX

Description

Specifies the string that shall be used as the current CAP_PRINTER device's suffix.

Application

Negotiate this capability to specify the string that is used as the suffix for printing if TWPM_COMPOUNDSTRING is used.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_STR255
Default Value:	No Default
Allowed Values:	Any string
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_PRINTER
CAP_PRINTERENABLED
CAP_PRINTERINDEX
CAP_PRINTERMODE
CAP_PRINTERSTRING

CAP_REWINDPAGE

Description

If TRUE, the Source will return the current page to the input side of the document feeder and feed the last page from the output side of the feeder back into the acquisition area.

If CAP_AUTOFEED is TRUE, automatic feeding will continue after all negotiated frames from this page are acquired.

CAP_FEEDERENABLED must equal TRUE to use this capability.

This capability must have been negotiated as an extended capability to be used in States 5 and 6.

Application

This capability is used in States 5 and 6 by applications controlling the Source's feeder (usually without the Source's user interface).

If CAP_AUTOFEED is TRUE, the normal automatic feeding will continue after all frames of this page are acquired.

Source

If CAP_FEEDERENABLED equals FALSE, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED (capability is not supported in current settings).

If there are no documents in the output area, return: TWRC_FAILURE / TWCC_BADVALUE.

The Source will perform this action once whenever the capability is MSG_SET to TRUE. The Source should then revert the Current value to FALSE.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_AUTOFEED
CAP_CLEARPAGE
CAP_EXTENDEDCAPS
CAP_FEEDERENABLED
CAP_FEEDERLOADED
CAP_FEEDPAGE

CAP_SERIALNUMBER

Description

A string containing the serial number of the currently selected device in the Source. Multiple devices may all report the same serial number.

Application

The value is device specific, Applications should not attempt to parse the information.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL /DAT_CAPABILITY/ MSG_QUERYSUPPORT)

Values

Type:	TW_STR255
Default Value:	No default
Allowed Values:	Any value
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

None

CAP_SUPPORTEDCAPS

Description

Returns a list of all the capabilities for which the Source will answer inquiries. Does not indicate which capabilities the Source will allow to be set by the application. Some capabilities can only be set if certain setup work has been done so the Source cannot globally answer which capabilities are "set-able."

Values

Type:	TW_UINT16
Default Value:	No Default
Allowed Values:	Any "get-able" capability
Container for MSG_GET:	TW_ARRAY
Container for MSG_SET:	MSG_SET not allowed

Required By

All Sources.

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

CAP_EXTENDEDCAPS

CAP_TIMEBEFOREFIRSTCAPTURE

Description

For automatic capture, this value selects the number of milliseconds before the first picture is to be taken, or the first image is to be scanned.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_INT32
Default Value:	0
Allowed Values:	0 or greater
Container for MSG_GET:	TW_ONEVALUE, TW_RANGE
Container for MSG_SET:	TW_ONEVALUE, TW_RANGE

Required By

None

Source Required Operations

None

See Also

CAP_AUTOMATICCAPTURE
CAP_TIMEBETWEENCAPTURES
CAP_XFERCOUNT

CAP_TIMEBETWEENCAPTURES

Description

For automatic capture, this value selects the milliseconds to wait between pictures taken, or images scanned.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_INT32
Default Value:	0
Allowed Values:	0 or greater
Container for MSG_GET:	TW_ONEVALUE, TW_RANGE
Container for MSG_SET:	TW_ONEVALUE, TW_RANGE

Required By

None

Source Required Operations

None

See Also

CAP_AUTOMATICCAPTURE
CAP_TIMEBEFOREFIRSTCAPTURE
CAP_XFERCOUNT

CAP_TIMEDATE

Description

The date and time the image was acquired.

Note: CAP_TIMEDATE does not return the *exact* time the image was acquired; rather, it returns the *closest available approximation* of the time the physical phenomena represented by the image was recorded. If the application needs the exact time of acquisition, the application should generate that value itself during the image acquisition procedure.

Stored in the form "YYYY/MM/DD HH:mm:SS.sss" where YYYY is the year, MM is the numerical month, DD is the numerical day, HH is the hour, mm is the minute, SS is the second, and sss is the millisecond.

This capability must be negotiated during State 7 before the call to the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER triplet. It must also be listed in the CAP_EXTENDEDCAPS capability by the data source.

Source

The time and date when the image was originally acquired (when the Source entered State 7).

Be sure to leave the space between the ending of the date and beginning of the time fields. Pad the unused characters after the string with zeros.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_STR32
Default Value:	No Default
Allowed Values:	Any date
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

None

See Also

CAP_AUTHOR
CAP_CAPTION

CAP_THUMBNAILSENABLED

Description

Allows an application to request the delivery of thumbnail representations for the set of images that are to be delivered.

Setting CAP_THUMBNAILSENABLED to TRUE turns on thumbnail mode. Images transferred thereafter will be sent at thumbnail size (exact thumbnail size is determined by the Data Source). Setting this capability to FALSE turns thumbnail mode off and returns full size images.

Application

A successful set of this capability to TRUE will cause the Source to deliver image thumbnails during normal data transfer operations. This mode remains in effect until this capability is set back to FALSE.

Source

A successful set of this capability to TRUE should enable the delivery of thumbnail images during normal data transfer. Setting this capability to FALSE will disable thumbnail delivery.

If not supported, return TWRC_FAILURE/ TWCC_CAPUNSUPPORTED.

Values

Type:	TW_BOOL
Default Value:	FALSE (do not deliver thumbnails).
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

All Image Store Data Sources.

Source Required Operations

MSG_GET, MSG_SET, MSG_GETCURRENT, MSG_RESET

See Also

ICAP_IMAGEDATASET

CAP_UICONTROLLABLE

Description

If TRUE, indicates that this Source supports acquisition with the UI disabled; i.e., TW_USERINTERFACE's ShowUI field can be set to FALSE. If FALSE, indicates that this Source can only support acquisition with the UI enabled.

Source

This capability was introduced in TWAIN 1.6. All Sources compliant with TWAIN 1.6 and above must support this capability. Sources that are not TWAIN 1.6-compliant may return TWRC_FAILURE / TWCC_BADCAP if they do not support this capability.

Application

A return value of TWRC_FAILURE / TWCC_CAPUNSUPPORTED indicates that the Source in use is not TWAIN 1.6-compliant. Therefore, the Source may ignore TW_USERINTERFACE's ShowUI field when MSG_ENABLEDS is issued. See the description of DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS for more details.

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

All Sources.

See Also

CAP_INDICATORS
DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

CAP_XFERCOUNT

Description

The application is willing to accept this number of images.

Application

Set this capability to the number of images you are willing to transfer per session. Common values are:

1	Application wishes to transfer only one image this session
-1	Application is willing to transfer multiple images

Source

If the application limits the number of images it is willing to receive, the Source should not make more transfers available than the specified number.

Values

Type:	TW_INT16
Default Value:	-1
Allowed Values:	-1 to 2^{15}
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

All Sources and applications

Source Required Operations

MSG_GET / CURRENT / DEFAULT,
MSG_SET / RESET

See Also

TW_PENDINGXFERS.Count

ICAP_AUTOMATICBORDERDETECTION

Description

Turns automatic border detection on and off.

Application

Negotiate this capability to determine the state of the AutoBorder detection.

ICAP_UNDEFINEDIMAGESIZE must be enabled for this feature to work.

Source

If supported, enable or disable automatic border detection according to the value specified. Default to FALSE for backward compatibility. For this capability to be enabled, ICAP_UNDEFINEDIMAGESIZE must be enabled.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_UNDEFINEDIMAGESIZE
ICAP_AUTOMATICDESKEW

ICAP_AUTOBRIGHT

Description

TRUE enables and FALSE disables the Source's Auto-brightness function (if any).

Source

If TRUE, apply auto-brightness function to acquired image before transfer.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_BRIGHTNESS

ICAP_AUTOMATICDESKEW

Description

Turns automatic deskew correction on and off.

Application

Negotiate this capability to enable or disable Automatic deskew.

Source

If supported, enable or disable the Automatic deskew feature according to the value specified for future transfers. Default to FALSE for backward compatibility. Some Sources may require ICAP_UNDEFINEDIMAGESIZE to be enabled.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_AUTOMATICBORDERDETECTION
ICAP_AUTOMATICROTATE
ICAP_UNDEFINEDIMAGESIZE

ICAP_AUTOMATICROTATE

Description

When TRUE this capability depends on intelligent features within the Source to automatically rotate the image to the correct position.

Application

If this capability is set to TRUE, then it must be assumed that no other correction is required (deskew, rotation, etc…); the Source is guaranteeing that it will deliver images in the correct orientation.

Source

There are no criteria for how this automatic rotation is determined. A Source may use a field of text, or some distinguishing non-text field, such as a barcode or a logo, or it may rely on form recognition to help rotate the document.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_AUTOMATICDESKEW
ICAP_ORIENTATION
ICAP_ROTATION

ICAP_BARCODEDETECTIONENABLED

Description

Turns bar code detection on and off.

Source

Support this capability if the scanner supports any Bar code recognition. If the device allows this feature to be turned off, then default to off. If the device does not support disabling this feature, report TRUE and disallow attempts to set FALSE.

If not supported, return TWRC_FAILURE/ TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_SUPPORTEDBARCODETYPES
ICAP_SUPPORTEDPATCHCODETYPES
ICAP_BARCODEMAXSEARCHPRIORITIES
ICAP_BARCODESEARCHPRIORITIES
ICAP_BARCODESEARCHMODE
ICAP_BARCODEMAXRETRIES
ICAP_BARCODETIMEOUT

ICAP_BARCODEMAXRETRIES

Description

Restricts the number of times a search will be retried if none are found on each page.

Application

Refine this capability to limit the number of times the bar code search algorithm is retried on a page that contains no bar codes.

Source

If supported, limit the number of retries the value specified.

If not supported, return TWRC_FAILURE/ TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT32
Default Value:	No Default
Allowed Values:	1 to $2^{32}-1$
Container for MSG_GET:	TW_ENUMERATION TW_RANGE TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_BARCODEDETECTIONENABLED
ICAP_SUPPORTEDBARCODETYPES
ICAP_SUPPORTEDPATCHCODETYPES
ICAP_BARCODEMAXSEARCHPRIORITIES
ICAP_BARCODESEARCHPRIORITIES
ICAP_BARCODESEARCHMODE
ICAP_BARCODETIMEOUT

ICAP_BARCODEMAXSEARCHPRIORITIES

Description

The maximum number of supported search priorities.

Application

Query this value to determine how many bar code detection priorities can be set.

Set this value to limit the number of priorities to speed the detection process.

Source

If bar code searches can be prioritized, report the maximum number of priorities allowed for a search.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT32
Default Value:	No Default
Allowed Values:	1 to 2^{32} –1
Container for MSG_GET:	TW_ENUMERATION TW_RANGE TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_BARCODEDETECTIONENABLED
ICAP_SUPPORTEDBARCODETYPES
ICAP_SUPPORTEDPATCHCODETYPES
ICAP_BARCODESEARCHPRIORITIES
ICAP_BARCODESEARCHMODE
ICAP_BARCODEMAXRETRIES
ICAP_BARCODETIMEOUT

ICAP_BARCODESEARCHMODE

Description

Restricts bar code searching to certain orientations, or prioritizes one orientation over the other.

Application

Negotiate this capability if the orientation of bar codes is already known to the application. Refinement of this capability can speed the bar code search.

Source

If set then apply the specified refinements to future bar code searches.

If not supported, return TWRC_FAILURE/ TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWBD_HORZ	0
	TWBD_VERT	1
	TWBD_HORZVERT	2
	TWBD_VERTHORZ	3
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_BARCODEDETECTIONENABLED
ICAP_SUPPORTEDBARCODETYPES
ICAP_SUPPORTEDPATCHCODETYPES
ICAP_BARCODEMAXSEARCHPRIORITIES
ICAP_BARCODESEARCHPRIORITIES
ICAP_BARCODEMAXRETRIES
ICAP_BARCODETIMEOUT

ICAP_BARCODESEARCHPRIORITIES

Description

A prioritized list of bar code types dictating the order in which bar codes will be sought.

Application

Set this capability to specify the order and priority for bar code searching. Refining the priorities to only the bar code types of interest to the application can speed the search process.

Source

If this type of search refinement is supported, then report the current values.

If set, then limit future searches to the specified bar codes in the specified priority order.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWBT_3OF9	0
	TWBT_2OF5INTERLEAVED	1
	TWBT_2OF5NONINTERLEAVED	2
	TWBT_CODE93	3
	TWBT_CODE128	4
	TWBT_UCC128	5
	TWBT_CODABAR	6
	TWBT_UPCA	7
	TWBT_UPCE	8
	TWBT_EAN8	9
	TWBT_EAN13	10
	TWBT_POSTNET	11
	TWBT_PDF417	12
	TWBT_2OF5INDUSTRIAL	13
	TWBT_2OF5MATRIX	14
	TWBT_2OF5DATALOGIC	15
	TWBT_2OF5IATA	16
	TWBT_3OF9FULLASCII	17
	TWBT_CODABARWITHSTARTSTOP	18
	TWBT_MAXICODE	19
Container for MSG_GET:	TW_ ARRAY	
Container for MSG_SET:	TW_ ARRAY	

Required By

None

Source Required Operations

None

See Also

ICAP_BARCODEDETECTIONENABLED
ICAP_SUPPORTEDBARCODETYPES
ICAP_SUPPORTEDPATCHCODETYPES
ICAP_BARCODEMAXSEARCHPRIORITIES
ICAP_BARCODESEARCHMODE
ICAP_BARCODEMAXRETRIES
ICAP_BARCODETIMEOUT

ICAP_BARCODETIMEOUT

Description

Restricts the total time spent on searching for a bar code on each page.

Application

Refine this value to tune the length of time the search algorithm is allowed to execute before giving up.

Source

If supported, limit the duration of a bar code search to the value specified.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT).

Values

Type:	TW_UINT32
Default Value:	No Default
Allowed Values:	1 to 2^{32} –1
Container for MSG_GET:	TW_ENUMERATION TW_RANGE TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_BARCODEDETECTIONENABLED
ICAP_SUPPORTEDBARCODETYPES
ICAP_SUPPORTEDPATCHCODETYPES
ICAP_BARCODEMAXSEARCHPRIORITIES
ICAP_BARCODESEARCHPRIORITIES
ICAP_BARCODESEARCHMODE
ICAP_BARCODEMAXRETRIES

ICAP_BITDEPTH

Description

Specifies the pixel bit depths for the Current value of ICAP_PIXELTYPE. For example, when using ICAP_PIXELTYPE = TWPT_GRAY, this capability specifies whether this is 8-bit gray or 4-bit gray.

This depth applies to all the data channels (for instance, the R, G, and B channels will all have this same bit depth for RGB data).

Application

The application should loop through all the ICAP_PIXELTYPEs it is interested in and negotiate the ICAP_BITDEPTH(s) for each.

For all allowed settings of ICAP_PIXELTYPE

- Set ICAP_PIXELTYPE
- Set ICAP_BITDEPTH for the current ICAP_PIXELTYPE

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If the bit depth in a MSG_SET is not supported for the current ICAP_PIXELTYPE setting, return TWRC_FAILURE / TWCC_BADVALUE.

Values

Type:	TW_UINT16
Default Value:	No Default
Allowed Values:	>=1
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

ICAP_PIXELTYPE

ICAP_BITDEPTHREDUCTION

Description

Specifies the Reduction Method the Source should use to reduce the bit depth of the data. Most commonly used with ICAP_PIXELTYPE = TWPT_BW to reduce gray data to black and white.

Application

Set the capability to the reduction method to be used in future acquisitions

Also select the Halftone or Threshold to be used.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWBR_THRESHOLD	0
	TWBR_HALFTONES	1
	TWBR_CUSTHALFTONE	2
	TWBR_DIFFUSION	3
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION	
	TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_CUSTHALFTONE
ICAP_HALFTONES
ICAP_PIXELTYPE
ICAP_THRESHOLD

ICAP_BITORDER

Description

Specifies how the bytes in an image are filled by the Source. TWBO_MSBFIRST indicates that the leftmost bit in the byte (usually bit 7) is the byte's Most Significant Bit.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16	
Default Value:	TWBO_MSBFIRST	
Allowed Values:	TWBO_LSBFIRST	0
	TWBO_MSBFIRST	1
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

ICAP_BITORDERCODES

ICAP_BITORDERCODES

Description

Used for CCITT data compression only. Indicates the bit order representation of the stored compressed codes.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16	
Default Value:	TWBO_LSBFIRST	
Allowed Values:	TWBO_LSBFIRST	0
	TWBO_MSBFIRST	1
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_COMPRESSION

ICAP_BRIGHTNESS

Description

The brightness values available within the Source.

Application

The application can use this capability to inquire, set, or restrict the values for BRIGHTNESS used in the Source.

Source

Source should normalize the values into the range. Make sure that a ‘0’ value is available as the Current Value when the Source starts up. If the Source’s ± range is asymmetric about the ‘0’ value, set range maxima to ±1000 and scale homogeneously from the ‘0’ value in each direction. This will yield a positive range whose step size differs from the negative range’s step size.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	0
Allowed Values:	-1000 to +1000
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_AUTOBRIGHT
ICAP_CONTRAST

ICAP_CCITTKFACTOR

Description

Used for CCITT Group 3 2-dimensional compression. The 'K' factor indicates how often the new compression baseline should be re-established. A value of 2 or 4 is common in facsimile communication. A value of zero in this field will indicate an infinite K factor—the baseline is only calculated at the beginning of the transfer.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values:

Type:	TW_UINT16
Default Value:	4
Allowed Values:	0 to 2^{16}
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_COMPRESSION

ICAP_COMPRESSION

Description

Allows the application and Source to identify which compression schemes they have in common for Buffered Memory and File transfers.

Note for File transfers:

Since only certain file formats support compression, this capability must be negotiated after setting the desired file format with ICAP_IMAGEFILEFORMAT.

TWCP_NONE	All Sources must support this.
TWCP_PACKBITS	Macintosh PackBits format, (can be used with TIFF or PICT)
TWCP_GROUP31D, TWCP_GROUP31DEOL, TWCP_GROUP32D, TWCP_GROUP4	Are all from the CCITT specification (now ITU), intended for document images (can be used with TIFF).
TWCP_JPEG	Intended for the compression of color photographs (can be used with TIFF, JFIF or SPIFF).
TWCP_LZW	A compression licensed by UNISYS (can be used with TIFF).
TWCP_JBIG	Intended for bitonal and grayscale document images (can be used with TIFF or SPIFF).
TWCP_PNG	This compression can only be used if ICAP_IMAGEFILEFORMAT is set to TWFF_PNG.
TWCP_RLE4, TWCP_RLE8, TWCP_BITFIELDS	These compressions can only be used if ICAP_IMAGEFILEFORMAT is set to TWFF_BMP.

Application

Applications must not assume that a Source can provide compressed Buffered Memory or File transfers, because many cannot. The application should use MSG_SET on a TW_ONEVALUE container to specify the compression type for future transfers.

Source

The current value of this setting specifies the compression method to be used in future transfers. If the image transfer mechanism is changed, then the allowed list must be modified to reflect the supported values. If the current value is not available on the new allowed list, then the Source must change it to its preferred value.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	TWCP_NONE	
Allowed Values:	TWCP_NONE	0
	TWCP_PACKBITS	1
	TWCP_GROUP31D	2
	TWCP_GROUP31DEOL	3
	TWCP_GROUP32D	4
	TWCP_GROUP4	5
	TWCP_JPEG	6
	TWCP_LZW	7
	TWCP_JBIG	8
	TWCP_PNG	9
	TWCP_RLE4	10
	TWCP_RLE8	11
	TWCP_BITFIELDS	12
Container for MSG_GET:	TW_ENUMERATION, TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION, TW_ONEVALUE	

Required By

All Image Sources.

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

DG_CONTROL / DAT_IMAGEMEMXFER / MSG_GET
DG_CONTROL / DAT_IMAGEFILEXFER / MSG_GET

CAP_XFERMECH
ICAP_IMAGEFILEFORMAT

ICAP_CONTRAST

Description

The contrast values available within the Source.

Application

The application can use this capability to inquire, set or restrict the values for CONTRAST used in the Source.

Source

Scale the values available internally into a homogeneous range between -1000 and 1000. Make sure that a '0' value is available as the Current value when the Source starts up. If the Source's ± range is asymmetric about the '0' value, set range maxima to ±1000 and scale homogeneously from the '0' value in each direction. This will yield a positive range whose step size differs from the negative range's step size.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	0
Allowed Values:	-1000 to +1000
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_BRIGHTNESS

ICAP_CUSTHALFTONE

Description

Specifies the square-cell halftone (dithering) matrix the Source should use to halftone the image.

Application

The application should also set ICAP_BITDEPTHREDUCTION to TWBR_CUSTHALFTONE to use this capability.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT8
Default Value:	No Default
Allowed Values:	Any rectangular array
Container for MSG_GET:	TW_ARRAY
Container for MSG_SET:	TW_ARRAY

Required By

None

Source Required Operations

None

See Also

ICAP_BITDEPTHREDUCTION

ICAP_EXPOSURETIME

Description

Specifies the exposure time used to capture the image, in seconds.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	>0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_FLASHUSED2
ICAP_LAMPSTATE
ICAP_LIGHTPATH
ICAP_LIGHTSOURCE

ICAP_EXTIMAGEINFO

Description

Allows the application to query the data source to see if it supports the new operation triplet DG_IMAGE / DAT_EXTIMAGEINFO / MSG_GET.

If TRUE, the source will support the DG_IMAGE / DAT_EXTIMAGEINFO / MSG_GET message.

Note: The TWAIN API allows for an application to query the results of many advanced device / manufacturer operations. The responsibility of configuring and setting up each advanced operation lies with the device's data source user interface. Since the configuration of advanced device / manufacturer-specific operations varies from manufacturer to manufacturer, placing the responsibility for setup and configuration of advanced operations allows the application to remain device independent.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

DG_IMAGE / DAT_EXTIMAGEINFO / MSG_GET

ICAP_FILTER

Description

Describes the color characteristic of the subtractive filter applied to the image data. Multiple filters may be applied to a single acquisition.

Source

If the Source only supports application of a single filter during an acquisition and multiple filters are specified by the application, set the current filter to the first one requested and return TWRC_CHECKSTATUS.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWFT_RED	0
	TWFT_GREEN	1
	TWFT_BLUE	2
	TWFT_NONE	3
	TWFT_WHITE	4
	TWFT_CYAN	5
	TWFT_MAGENTA	6
	TWFT_YELLOW	7
	TWFT_BLACK	8
Container for MSG_GET:	TW_ARRAY	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ARRAY	
	TW_ONEVALUE	

Required By

None

Source Required Operations

None

ICAP_FLASHUSED

Description

Specifies whether or not the image was acquired using a flash.

Application

Note that an image with flash may have a different color composition than an image without flash.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_EXPOSURETIME
ICAP_FLASHUSED2
ICAP_LAMPSTATE
ICAP_LIGHTPATH
ICAP_LIGHTSOURCE

ICAP_FLASHUSED2

Description

For devices that support flash. MSG_SET selects the flash to be used (if any). MSG_GET reports the current setting. This capability replaces ICAP_FLASHUSED, which is only able to negotiate the flash being on or off.

Application

Note that an image with flash may have a different color composition than an image without flash.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	TWFL_NONE	
Allowed Values:	TWFL_NONE	0
	TWFL_OFF	1
	TWFL_ON	2
	TWFL_AUTO	3
	TWFL_REDEYE	4
Container for MSG_GET:	TW_ENUMERATION, TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION, TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

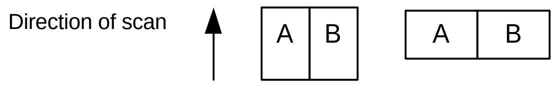
ICAP_FLASHUSED

ICAP_FLIPROTATION

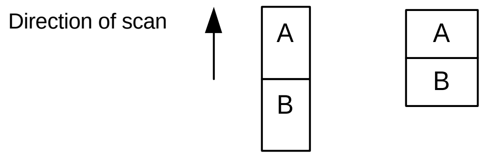
Description

Flip rotation is used to properly orient images that flip orientation every other image.

TWFR_BOOK	The images to be scanned are viewed in book form, flipping each page from left to right or right to left.



TWFR_FANFOLD	The images to be scanned are viewed in fanfold paper style, flipping each page up or down.



On duplex paper, the As are all located on the top, and the Bs are all located on the bottom. If ICAP_FLIPROTATION is set to TWFR_BOOK, and fanfold paper is scanned, then every B image will be upside down. Setting the capability to TWFR_FANFOLD instructs the Source to rotate the B images 180 degrees around the x-axis.

Because this capability is described to act upon every other image, it will work correctly in simplex mode, assuming that every other simplex image is flipped in the manner described above.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	TWFR_BOOK	
Allowed Values:	TWFR_BOOK	0
	TWFR_FANFOLD	1
Container for MSG_GET:	TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

None

Source Required Operations

None

ICAP_FRAMES

Description

The list of frames the Source will acquire on each page.

Application

MSG_GET returns the size and location of all the frames the Source will acquire image data from when acquiring from each page.

MSG_GETCURRENT returns the size and location of the next frame to be acquired.

MSG_SET allows the application to specify the frames and their locations to be used to acquire from future pages.

This ICAP is most useful if the Source supports simultaneous acquisition from multiple frames. Use ICAP_MAXFRAMES to establish this ability.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FRAME
Default Value:	No Default
Allowed Values:	Device dependent
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_MAXFRAMES
ICAP_SUPPORTEDSIZES
TW_IMAGELAYOUT

ICAP_GAMMA

Description

Gamma correction value for the image data.

Application

Do not use with TW_CIECOLOR, TW_GRAYRESPONSE, or TW_RGBRESPONSE data.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	2.2
Allowed Values:	Any value
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

ICAP_HALFTONES

Description

A list of names of the halftone patterns available within the Source.

Application

The application may not rename any halftone pattern.

The application should also set ICAP_BITDEPTHREDUCTION to use this capability.

Values

Type:	TW_STR32
Default Value:	No Default
Allowed Values:	Any halftone name
Container for MSG_GET:	TW_ARRAY (for backwards compatibility with 1.0 only) TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	TW_ARRAY (for backwards compatibility with 1.0 only) TW_ENUMERATION TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_CUSTHALFTONE
ICAP_BITDEPTHREDUCTION
ICAP_THRESHOLD

ICAP_HIGHLIGHT

Description

Specifies which value in an image should be interpreted as the lightest "highlight." All values "lighter" than this value will be clipped to this value. Whether lighter values are smaller or larger can be determined by examining the Current value of ICAP_PIXELFLAVOR.

Source

If more or less than 8 bits are used to describe the image, the actual data values should be normalized to fit within the 0-255 range. The normalization need not result in a homogeneous distribution if the original distribution was not homogeneous.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	255
Allowed Values:	0 to 255
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_SHADOW

ICAP_IMAGEDATASET

Description

Gets or sets the image indices that will be delivered during the standard image transfer done in States 6 and 7. Indices are assumed to start at 1, so a TW_ONEVALUE container sets an implied range from 1 to the number specified. TW_RANGE returns are useful for those cases where the images are contiguous (5 .. 36). TW_ARRAY returns should be used were index values are discontinuous (as could be the case where the user previously set such a data set). See the note in the Values section below.

Application

A MSG_RESET operation should always be done before a MSG_GET if the application wishes to get the complete list of available images. A MSG_SET operation will define the number and order of images delivered during States 6 and 7.

Source

For MSG_GET, if a contiguous range of images are available starting from the first index (e.g., 1 .. 36) it is recommended that the TW_ONEVALUE container is used specifying just the total number of available images (e.g., 36).

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT32
Default Value:	Entire range or set of available images
Allowed Values:	0 to 2^{32} -1 (for MSG_GET) 1 to 2^{32} -1 (for MSG_SET)
Container for MSG_GET:	TW_ONEVALUE TW_RANGE (see note below) TW_ARRAY (see note below)
Container for MSG_SET:	TW_ONEVALUE TW_RANGE TW_ARRAY

Note: These container types are supported for the returning discontinuous indices that have been previously set by the application. It is highly recommended that for a initialized or reset *Image Store* device, the TW_ONEVALUE container be the only one returned by the MSG_GET operation. In other words, the data source should not expose the details of the internal memory management of the Image Store device by claiming that it has a hole in its storage locations due to user deletions. For example, a camera that currently has data for pictures 1 to 10 should report that it has 10 images available. If the user later deletes pictures 5, 7, and 9, it should now report that it has 7 images available (i.e., 1 to 7), and not claim that it has pictures 1, 2, 3, 4, 6, 8, and 10 available. To do so would expose the internal memory management constraints of the device and serves little use but to confuse the user.

Required By

All *Image Store* Data Sources.

Source Required Operations

MSG_GET, MSG_SET, MSG_RESET

ICAP_IMAGEFILEFORMAT

Description

Informs the application which file formats the Source can generate (MSG_GET). Tells the Source which file formats the application can handle (MSG_SET).

TWFF_TIFF	Used for document imaging
TWFF_PICT	Native Macintosh format
TWFF_BMP	Native Microsoft format
TWFF_XBM	Used for document imaging
TWFF_JFIF	Wrapper for JPEG images
TWFF_FPX	FlashPix, used with digital cameras
TWFF_TIFFMULTI	Multi-page TIFF files
TWFF_PNG	An image format standard intended for use on the web, replaces GIF
TWFF_SPIFF	A standard from JPEG, intended to replace JFIF, also supports JBIG
TWFF_EXIF	

Application

Use this ICAP to determine which formats are available for file transfers, and set the context for other capability negotiations such as ICAP_COMPRESSION.

Be sure to use the DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation to specify the format to be used for a particular acquisition.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	TWFF_BMP (Windows) TWFF_PICT (Macintosh)	
Allowed Values:	TWFF_TIFF	0
	TWFF_PICT	1
	TWFF_BMP	2
	TWFF_XBM	3
	TWFF_JFIF	4
	TWFF_FPX	5
	TWFF_TIFFMULTI	6
	TWFF_PNG	7

	TWFF_SPIFF	8
	TWFF_EXIF	9
Container for MSG_GET:	TW_ENUMERATION, TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION, TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET
DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET

ICAP_COMPRESSION

ICAP_IMAGEFILTER

Description

For devices that support image enhancement filtering. This capability selects the algorithm used to improve the quality of the image.

Application

TWIF_LOWPASS is good for halftone images.

TWIF_BANDPASS is good for improving text.

TWIF_HIGHPASS is good for improving fine lines.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	TWIF_NONE	
Allowed Values:	TWIF_NONE	0
	TWIF_AUTO	1
	TWIF_LOWPASS	2
	TWIF_BANDPASS	3
	TWIF_HIGHPASS	4
Container for MSG_GET:	TW_ENUMERATION, TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION, TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_JPEGPIXELTYPE

Description

Allows the application and Source to agree upon a common set of color descriptors that are made available by the Source. This ICAP is only useful for JPEG-compressed buffered memory image transfers.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWPT_BW	0
	TWPT_GRAY	1
	TWPT_RGB	2
	TWPT_PALETTE	3
	TWPT_CMY	4
	TWPT_CMYK	5
	TWPT_YUV	6
	TWPT_YUVK	7
	TWPT_CIEXYZ	8
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION	
	TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_COMPRESSION

ICAP_LAMPSTATE

Description

TRUE means the lamp is currently, or should be set to ON. Sources may not support MSG_SET operations.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_EXPOSURETIME
ICAP_FLASHUSED2
ICAP_LIGHTPATH
ICAP_LIGHTSOURCE

ICAP_LIGHTPATH

Description

Describes whether the image was captured transmissively or reflectively.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWLP_REFLECTIVE	0
	TWLP_TRANSMISSIVE	1
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_EXPOSURETIME
ICAP_FLASHUSED2
ICAP_LAMPSTATE
ICAP_LIGHTSOURCE

ICAP_LIGHTSOURCE

Description

Describes the general color characteristic of the light source used to acquire the image.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWLS_RED	0
	TWLS_GREEN	1
	TWLS_BLUE	2
	TWLS_NONE	3
	TWLS_WHITE	4
	TWLS_UV	5
	TWLS_IR	6
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION	
	TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_EXPOSURETIME
ICAP_FLASHUSED2
ICAP_LAMPSTATE
ICAP_LIGHTPATH

ICAP_MAXFRAMES

Description

The maximum number of frames the Source can provide or the application can accept per page.

This is a bounding capability only. It does not establish current or future behavior.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16
Default Value:	No Default
Allowed Values:	1 to 2^{16}
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_FRAMES
TW_IMAGELAYOUT

ICAP_MINIMUMHEIGHT

Description

Allows the source to define the minimum height (Y-axis) that the source can acquire.

Application

Source

The minimum height that the device can scan. This may be different depending on the value of CAP_FEEDERENABLED.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	0 to 32767 in ICAP_UNITS
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

See Also

CAP_FEEDERENABLED
ICAP_PHYSICALHEIGHT
ICAP_UNITS

ICAP_MINIMUMWIDTH

Description

Allows the source to define theminimum width (X-axis) that the source can acquire.

Source

The minimum width that the device can scan. This may be different depending on the value of CAP_FEEDERENABLED.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	0 to 32767 in ICAP_UNITS
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

See Also

CAP_FEEDERENABLED
ICAP_PHYSICALWIDTH
ICAP_UNITS

ICAP_NOISEFILTER

Description

For devices that support noise filtering. This capability selects the algorithm used to remove noise.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	TWNF_NONE	
Allowed Values:	TWNF_NONE	0
	TWNF_AUTO	1
	TWNF_LONEPIXEL	2
	TWNF_MAJORITYRULE	3
Container for MSG_GET:	TW_ENUMERATION, TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION, TW_ONEVALUE	

Required By

None

Source Required Operations

None

ICAP_ORIENTATION

Description

Defines which edge of the “paper” the image’s “top” is aligned with. This information is used to adjust the frames to match the scanning orientation of the paper. For instance, if an ICAP_SUPPORTEDSIZE of TWSS_ISOA4 has been negotiated, and ICAP_ORIENTATION is set to TWOR_LANDSCAPE, then the Source must rotate the frame it downloads to the scanner to reflect the orientation of the paper. Please note that setting ICAP_ORIENTATION does not affect the values reported by ICAP_FRAMES; it just causes the Source to use them in a different way.

The upper-left of the image is defined as the location where both the primary and secondary scans originate. (The X axis is the primary scan direction and the Y axis is the secondary scan direction.) For a flatbed scanner, the light bar moves in the secondary scan direction. For a handheld scanner, the scanner is drug in the secondary scan direction. For a digital camera, the secondary direction is the vertical axis when the viewed image is considered upright.

Application

If one pivots the image about its center, then orienting the image in TWOR_LANDSCAPE has the effect of rotating the original image 90 degrees to the “left.” TWOR_PORTRAIT mode does not rotate the image. The image may be oriented along any of the four axes located 90 degrees from the unrotated image. Note that:

TWOR_ROT0 == TWOR_PORTRAIT and TWOR_ROT270 == TWOR_LANDSCAPE.

Source

The Source is responsible for rotating the image if it allows this capacity to be set.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16	
Default Value:	TWOR_PORTRAIT	
Allowed Values:	TWOR_ROT0	0
	TWOR_ROT90	1
	TWOR_ROT180	2
	TWOR_ROT270	3
	TWOR_PORTRAIT	(equals TWOR_ROT0)
	TWOR_LANDSCAPE	(equals TWOR_ROT270)
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION	
	TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

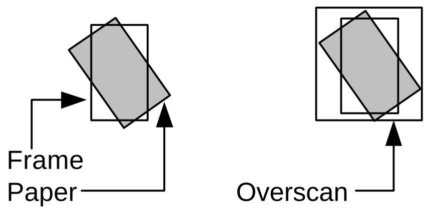
ICAP_ROTATION

ICAP_OVERSCAN

Description

Overscan is used to scan outside of the boundaries described by ICAP_FRAMES, and is used to help acquire image data that may be lost because of skewing.

Consider the following:



This is primarily of use for transport scanners which rely on edge detection to begin scanning. If overscan is supported, then the device is capable of scanning in the inter-document gap to get the skewed image information.

Application

Use this capability, if available, to help software processing images for deskew and border removal.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	TWOV_NONE	
Allowed Values:	TWOV_NONE	0
	TWOV_AUTO	1
	TWOV_TOPBOTTOM	2
	TWOV_LEFTRIGHT	3
	TWOV_ALL	4
Container for MSG_GET:	TW_ENUMERATION, TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION, TW_ONEVALUE	

Required By

None

Source Required Operations

None

ICAP_PATCHCODEDETECTIONENABLED

Description

Turns patch code detection on and off.

Source

Support this capability if the scanner supports any patch code recognition. If the device allows this feature to be turned off, then default to off. If the device does not support disabling this feature, report TRUE and disallow attempts to set FALSE.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_SUPPORTEDPATCHCODETYPES
ICAP_PATCHCODEMAXSEARCHPRIORITIES
ICAP_PATCHCODESEARCHPRIORITIES
ICAP_PATCHCODESEARCHMODE
ICAP_PATCHCODEMAXRETRIES
ICAP_PATCHCODETIMEOUT

ICAP_PATCHCODEMAXRETRIES

Description

Restricts the number of times a search will be retried if none are found on each page.

Application

Refine this capability to limit the number of times the patch code search algorithm is retried on a page that contains no patch codes.

Source

If supported, limit the number of retries the value specified.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT32
Default Value:	No Default
Allowed Values:	1 to 2^{32} –1
Container for MSG_GET:	TW_ENUMERATION TW_RANGE TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_PATCHCODEDETECTIONENABLED
ICAP_SUPPORTEDPATCHCODETYPES
ICAP_PATCHCODEMAXSEARCHPRIORITIES
ICAP_PATCHCODESEARCHPRIORITIES
ICAP_PATCHCODESEARCHMODE
ICAP_PATCHCODETIMEOUT

ICAP_PATCHCODEMAXSEARCHPRIORITIES

Description

The maximum number of supported search priorities.

Application

Query this value to determine how many patch code detection priorities can be set.

Source

Set this value to limit the number of priorities to speed the detection process.

If patch code searches can be prioritized, report the maximum number of priorities allowed for a search.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT32
Default Value:	No Default
Allowed Values:	1 to 2^{32} –1
Container for MSG_GET:	TW_ENUMERATION TW_RANGE TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_PATCHCODEDETECTIONENABLED
ICAP_SUPPORTEDPATCHCODETYPES
ICAP_PATCHCODESEARCHPRIORITIES
ICAP_PATCHCODESEARCHMODE
ICAP_PATCHCODEMAXRETRIES
ICAP_PATCHCODETIMEOUT

ICAP_PATCHCODESEARCHMODE

Description

Restricts patch code searching to certain orientations, or prioritizes one orientation over the other.

Application

Negotiate this capability if the orientation of patch codes is already known to the application. Refinement of this capability can speed the patch code search.

Source

If set then apply the specified refinements to future patch code searches.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWBD_HORZ	0
	TWBD_VERT	1
	TWBD_HORZVERT	2
	TWBD_VERTHORZ	3
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_PATCHCODEDETECTIONENABLED
ICAP_SUPPORTEDPATCHCODETYPES
ICAP_PATCHCODEMAXSEARCHPRIORITIES
ICAP_PATCHCODESEARCHPRIORITIES
ICAP_PATCHCODEMAXRETRIES
ICAP_PATCHCODETIMEOUT

ICAP_PATCHCODESEARCHPRIORITIES

Description

A prioritized list of patch code types dictating the order in which patch codes will be sought.

Application

Set this capability to specify the order and priority for patch code searching. Refining the priorities to only the patch code types of interest to the application can speed the search process.

Source

If this type of search refinement is supported, then report the current values.

If set, then limit future searches to the specified patch codes in the specified priority order.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16
Default Value:	No Default
Allowed Values:	TWPCH_PATCH1 TWPCH_PATCH2 TWPCH_PATCH3 TWPCH_PATCH4 TWPCH_PATCH6 TWPCH_PATCHT
Container for MSG_GET:	TW_ARRAY
Container for MSG_SET:	TW_ ARRAY

Required By

None

Source Required Operations

None

See Also

ICAP_PATCHCODEDETECTIONENABLED
ICAP_SUPPORTEDPATCHCODETYPES
ICAP_PATCHCODEMAXSEARCHPRIORITIES
ICAP_PATCHCODESEARCHMODE
ICAP_PATCHCODEMAXRETRIES
ICAP_PATCHCODETIMEOUT

ICAP_PATCHCODETIMEOUT

Description

Restricts the total time spent on searching for a patch code on each page.

Application

Refine this value to tune the length of time the search algorithm is allowed to execute before giving up.

Source

If supported, limit the duration of a patch code search to the value specified.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT32
Default Value:	No Default
Allowed Values:	1 to 2^{32} –1
Container for MSG_GET:	TW_ENUMERATION TW_RANGE TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_PATCHCODEDETECTIONENABLED
ICAP_SUPPORTEDPATCHCODETYPES
ICAP_PATCHCODEMAXSEARCHPRIORITIES
ICAP_PATCHCODESEARCHPRIORITIES
ICAP_PATCHCODESEARCHMODE
ICAP_PATCHCODEMAXRETRIES

ICAP_PHYSICALHEIGHT

Description

The maximum physical height (Y-axis) the Source can acquire (measured in units of ICAP_UNITS).

Source

For a flatbed scanner, the scannable height of the platen. For a handheld scanner, the maximum length of a scan.

For dimensionless devices, such as digital cameras, this ICAP is meaningless for all values of ICAP_UNITS other than TWUN_PIXELS. If the device is dimensionless, the Source should return a value of zero if ICAP_UNITS does not equal TWUN_PIXELS. This tells the application to inquire with TWUN_PIXELS.

Note: The physical acquired area may be different depending on the setting of CAP_FEEDERENABLED (if the Source has separate feeder and non-feeder acquire areas).

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	0 to 65535 in ICAP_UNITS
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

CAP_FEEDERENABLED
ICAP_UNITS

ICAP_PHYSICALWIDTH

Description

The maximum physical width (X-axis) the Source can acquire (measured in units of ICAP_UNITS).

Source

For a flatbed scanner, the scannable width of the platen. For a handheld scanner, the maximum width of a scan.

For dimensionless devices, such as digital cameras, this ICAP is meaningless for all values of ICAP_UNITS other than TWUN_PIXELS. If the device is dimensionless, the Source should return a value of zero if ICAP_UNITS does not equal TWUN_PIXELS. This tells the application to inquire with TWUN_PIXELS. The Source should then reply with its X-axis pixel count.

Note: The physical acquired area may be different depending on the setting of CAP_FEEDERENABLED (if the Source has separate feeder and non-feeder acquire areas).

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	0 to 65535 in ICAP_UNITS
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

CAP_FEEDERENABLED
ICAP_UNITS

ICAP_PIXELFLAVOR

Description

Sense of the pixel whose numeric value is zero (minimum data value). For example, consider a black and white image:

```
If ICAP_PIXELTYPE is TWPT_BW then
  If ICAP_PIXELFLAVOR is TWPF_CHOCOLATE
      then Black = 0
  Else if ICAP_PIXELFLAVOR is TWPF_VANILLA
      then White = 0
```

Application

Sources may prefer a different value depending on ICAP_PIXELTYPE. Set ICAP_PIXELTYPE and do a MSG_GETDEFAULT to determine the Source's preferences.

Source

TWPF_CHOCOLATE means this pixel represents the darkest data value that can be generated by the device (the darkest available optical value may measure greater than 0).

TWPF_VANILLA means this pixel represents the lightest data value that can be generated by the device (the lightest available optical value may measure greater than 0).

Values

Type:	TW_UINT16	
Default Value:	TWPF_CHOCOLATE	
Allowed Values:	TWPF_CHOCOLATE	0
	TWPF_VANILLA	1
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

All Image Sources

Source Required Operations

MSG_GET / CURRENT / DEFAULT,
MSG_SET / RESET

See Also

ICAP_PIXELTYPE

ICAP_PIXELFLAVORCODES

Description

Used only for CCITT data compression. Specifies whether the compressed codes' pixel "sense" will be inverted from the Current value of ICAP_PIXELFLAVOR prior to transfer.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16	
Default Value:	TWPF_CHOCOLATE	
Allowed Values:	TWPF_CHOCOLATE	0
	TWPF_VANILLA	1
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_COMPRESSION

ICAP_PIXELTYPE

Description

The type of pixel data that a Source is capable of acquiring (for example, black and white, gray, RGB, etc.).

Application

- MSG_GET returns a list of all pixel types available from the Source.
- MSG_SET on a TW_ENUMERATION structure requests that the Source restrict the available pixel types to the enumerated list.
- MSG_SET on a TW_ONEVALUE container specifies the only pixel type the application can accept.

If the application plans to transfer data through any mechanism other than Native and cannot handle all possible ICAP_PIXELTYPEs, it **must** support negotiation of this ICAP.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWPT_BW	0
	TWPT_GRAY	1
	TWPT_RGB	2
	TWPT_PALETTE	3
	TWPT_CMY	4
	TWPT_CMYK	5
	TWPT_YUV	6
	TWPT_YUVK	7
	TWPT_CIEXYZ	8
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION	
	TW_ONEVALUE	

Required By

All Image Sources

Source Required Operations

MSG_GET / CURRENT / DEFAULT,
MSG_SET / RESET

See Also

ICAP_BITDEPTH
ICAP_BITDEPTHREDUCTION

ICAP_PLANARCHUNKY

Description

Allows the application and Source to identify which color data formats are available. There are two options, “planar” and “chunky.”

For example, planar RGB data is transferred with the entire red plane of data first, followed by the entire green plane, followed by the entire blue plane (typical for three-pass scanners). “Chunky” mode repetitively interlaces a pixel from each plane until all the data is transferred (R-G-B-R-G-B…) (typical for one-pass scanners).

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWPC_CHUNKY	0
	TWPC_PLANAR	1
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION	
	TW_ONEVALUE	

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

TW_IMAGEINFO.Planar

ICAP_ROTATION

Description

How the Source can/should rotate the scanned image data prior to transfer. This doesn't use ICAP_UNITS. It is always measured in degrees. Any applied value is additive with any rotation specified in ICAP_ORIENTATION.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	0
Allowed Values:	+/- 360 degrees
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_ORIENTATION

ICAP_SHADOW

Description

Specifies which value in an image should be interpreted as the darkest “shadow.” All values “darker” than this value will be clipped to this value.

Application

Whether darker values are smaller or larger can be determined by examining the Current value of ICAP_PIXELFLAVOR.

Source

If more or less than 8 bits are used to describe the image, the actual data values should be normalized to fit within the 0-255 range. The normalization need not result in a homogeneous distribution if the original distribution was not homogeneous.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	0
Allowed Values:	0 to 255
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_PIXELFLAVOR

ICAP_SUPPORTEDBARCODETYPES

Description

Provides a list of bar code types that can be detected by the current Data Source.

Application

Query this capability to determine if the Data Source can detect bar codes that are appropriate to the particular application.

Source

If bar code detection is supported, report all the bar code types that can be detected.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWBT_3OF9	0
	TWBT_2OF5INTERLEAVED	1
	TWBT_2OF5NONINTERLEAVED	2
	TWBT_CODE93	3
	TWBT_CODE128	4
	TWBT_UCC128	5
	TWBT_CODABAR	6
	TWBT_UPCA	7
	TWBT_UPCE	8
	TWBT_EAN8	9
	TWBT_EAN13	10
	TWBT_POSTNET	11
	TWBT_PDF417	12
	TWBT_2OF5INDUSTRIAL	13
	TWBT_2OF5MATRIX	14
	TWBT_2OF5DATALOGIC	15
	TWBT_2OF5IATA	16
	TWBT_3OF9FULLASCII	17
	TWBT_CODABARWITHSTARTSTOP	18
	TWBT_MAXICODE	19
Container for MSG_GET:	TW_ARRAY	
Container for MSG_SET:	MSG_SET not allowed	

Required By

None

Source Required Operations

None

See Also

ICAP_BARCODEDETECTIONENABLED
ICAP_SUPPORTEDPATCHCODETYPES
ICAP_BARCODEMAXSEARCHPRIORITIES
ICAP_BARCODESEARCHPRIORITIES
ICAP_BARCODESEARCHMODE
ICAP_BARCODEMAXRETRIES
ICAP_BARCODETIMEOUT

ICAP_SUPPORTEDPATCHCODETYPES

Description

A list of patch code types that may be detected by the current Data Source.

Application

Query this capability to determine if the Data Source can detect patch codes that are appropriate to the Application.

Source

If patch code detection is supported, report all the possible patch code types that might be detected.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16
Default Value:	No Default
Allowed Values:	TWPCH_PATCH1 TWPCH_PATCH2 TWPCH_PATCH3 TWPCH_PATCH4 TWPCH_PATCH6 TWPCH_PATCHT
Container for MSG_GET:	TW_ARRAY TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

None

See Also

ICAP_PATCHCODEDETECTIONENABLED
ICAP_PATCHCODEMAXSEARCHPRIORITIES
ICAP_PATCHCODESEARCHPRIORITIES
ICAP_PATCHCODESEARCHMODE
ICAP_PATCHCODEMAXRETRIES
ICAP_PATCHCODETIMEOUT

ICAP_SUPPORTEDSIZES

Description

For devices that support fixed frame sizes. Defined sizes match typical page sizes. This specifies the size(s) the Source can / should use to acquire image data.

(*) Constant should not be used in Sources or Applications using TWAIN 1.8 or higher. For instance, use TWSS_A4 instead of TWSS_A4LETTER (note that the values are the same, the reason for the new constants is to improve naming clarification and consistency).

Note: TWSS_B has been removed from the specification.

Source

The frame size selected by using this capability should be reflected in the TW_IMAGELAYOUT structure information.

If the Source cannot acquire the exact frame size specified by the application, it should provide the closest possible size (preferably acquiring an image that is larger than the requested frame in both axes).

For devices that support physical dimensions TWSS_NONE indicates that the maximum image size supported by the device is to be used. Devices that do not support physical dimensions should not support this capability.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWSS_NONE	0
	*TWSS_A4LETTER	1
	*TWSS_B5LETTER	2
	TWSS_USLETTER	3
	TWSS_USLEGAL	4
	TWSS_A5	5
	*TWSS_B4	6
	*TWSS_B6	7
	TWSS_USLEDGER	9
	TWSS_USEXECUTIVE	10
	TWSS_A3	11
	*TWSS_B3	12
	*TWSS_A6	13
	TWSS_C4	14

TWSS_C5	15
TWSS_C6	16
// 1.8 Additions	
TWSS_4A0	17
TWSS_2A0	18
TWSS_A0	19
TWSS_A1	20
TWSS_A2	21
TWSS_A4	TWSS_A4LETTER
TWSS_A7	22
TWSS_A8	23
TWSS_A9	24
TWSS_A10	25
TWSS_ISOB0	26
TWSS_ISOB1	27
TWSS_ISOB2	28
TWSS_ISOB3	TWSS_B3
TWSS_ISOB4	TWSS_B4
TWSS_ISOB5	29
TWSS_ISOB6	TWSS_B6
TWSS_ISOB7	30
TWSS_ISOB8	31
TWSS_ISOB9	32
TWSS_ISOB10	33
TWSS_JISB0	34
TWSS_JISB1	35
TWSS_JISB2	36
TWSS_JISB3	37
TWSS_JISB4	38
TWSS_JISB5	TWSS_B5LETTER
TWSS_JISB6	39
TWSS_JISB7	40
TWSS_JISB8	41
TWSS_JISB9	42
TWSS_JISB10	43
TWSS_C0	44
TWSS_C1	45
TWSS_C2	46
TWSS_C3	47
TWSS_C7	48
TWSS_C8	49
TWSS_C9	50
TWSS_C10	51
TWSS_USSTATEMENT	52
TWSS_BUSINESSCARD	53

Container for MSG_GET: TW_ENUMERATION,
TW_ONEVALUE

Container for MSG_SET: TW_ENUMERATION,
TW_ONEVALUE

Required By

All Image Sources that support fixed frame sizes.

Source Required Operations

MSG_GET / CURRENT / DEFAULT, MSG_SET / RESET

See Also

ICAP_FRAMES
TW_IMAGEINFO
TW_IMAGELAYOUT

ICAP_THRESHOLD

Description

Specifies the dividing line between black and white. This is the value the Source will use to threshold, if needed, when ICAP_PIXELTYPE = TWPT_BW.

The value is normalized so there are no units of measure associated with this ICAP.

Application

Application will typically set ICAP_BITDEPTHREDUCTION to TWBR_THRESHOLD to use this capability.

Source

Source should fit available values linearly into the defined range such that the lowest available value equals 0 and the highest equals 255.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	128
Allowed Values:	0 to 255
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_BITDEPTHREDUCTION

ICAP_TILES

Description

This is used with buffered memory transfers. If TRUE, Source can provide application with tiled image data.

Application

If set to TRUE, the application expects the Source to supply tiled data for the upcoming transfer(s). This persists until the application sets it to FALSE. If the application sets it to FALSE, Source will supply strip data.

Source

If Source can supply tiled data and application does not set this ICAP, Source may or may not supply tiled data at its discretion.

In State 6, ICAP_TILES should reflect whether tiles or strips will be used in the upcoming transfer.

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

TW_IMAGEMEMXFER

ICAP_TIMEFILL

Description

Used only with CCITT data compression. Specifies the minimum number of words of compressed codes (compressed data) to be transmitted per line.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16
Default Value:	1
Allowed Values:	1 to 2^{16}
Container for MSG_GET:	TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_COMPRESSION

ICAP_UNDEFINEDIMAGESIZE

Description

If TRUE the Source will issue a MSG_XFERRDY before starting the scan.

Note: The Source may need to scan the image before initiating the transfer. This is the case if the scanned image is rotated or merged with another scanned image.

Application

Used by the application to notify the Source that the application accepts -1 as the image width or -length in the TW_IMAGEINFO structure.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

TW_IMAGEINFO

ICAP_UNITS

Description

Unless a quantity is dimensionless or uses a specified unit of measure, ICAP_UNITS determines the unit of measure for all quantities.

Application

Applications should be able to handle TWUN_PIXELS if they want to support data transfers from “dimensionless” devices such as digital cameras.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_UINT16	
Default Value:	TWUN_INCHES	
Allowed Values:	TWUN_INCHES	0
	TWUN_CENTIMETERS	1
	TWUN_PICAS	2
	TWUN_POINTS	3
	TWUN_TWIPS	4
	TWUN_PIXELS	5
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION	
	TW_ONEVALUE	

Required By

All Image Sources

Source Required Operations

MSG_GET / CURRENT / DEFAULT,
MSG_SET / RESET

See Also

ICAP_FRAMES
DAT_IMAGELAYOUT

ICAP_XFERMECH

Description

Allows the application and Source to identify which transfer mechanisms they have in common.

Application

The current setting of ICAP_XFERMECH must match the constant used by the application to specify the transfer mechanism when starting the transfer using the triplet: DG_IMAGE / DAT_IMAGExxxxXFER / MSG_GET.

Values

Type:	TW_UINT16	
Default Value:	TWSX_NATIVE	
Allowed Values:	TWSX_NATIVE	0
	TWSX_FILE	1
	TWSX_MEMORY	2
Container for MSG_GET:	TW_ENUMERATION	
	TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION	
	TW_ONEVALUE	

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT,
MSG_SET/RESET

See Also

DG_IMAGE / DAT_IMAGExxxxXFER / MSG_GET

ICAP_XNATIVERESOLUTION

Description

The native optical resolution along the X-axis of the device being controlled by the Source. Most devices will respond with a single value (TW_ONEVALUE).

This is NOT a list of all resolutions that can be generated by the device. Rather, this is the resolution of the device's optics. Measured in units of pixels per unit as defined by ICAP_UNITS (pixels per TWUN_PIXELS yields dimensionless data).

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	>0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

None

See Also

ICAP_UNITS
ICAP_XRESOLUTION
ICAP_YNATIVERESOLUTION

ICAP_XRESOLUTION

Description

All the X-axis resolutions the Source can provide.

Measured in units of pixels per unit as defined by ICAP_UNITS (pixels per TWUN_PIXELS yields dimensionless data). That is, when the units are TWUN_PIXELS, both ICAP_XRESOLUTION and ICAP_YRESOLUTION shall report 1 pixel/pixel. Some data sources like to report the actual number of pixels that the device reports, but that response is more appropriate in ICAP_PHYSICALHEIGHT and ICAP_PHYSICALWIDTH.

Application

Setting this value will restrict the various resolutions that will be available to the user during acquisition.

Applications will want to ensure that the values set for this ICAP match those set for ICAP_YRESOLUTION.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	>0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT,
MSG_SET/RESET

See Also

ICAP_UNITS
ICAP_XNATIVERESOLUTION
ICAP_YRESOLUTION

ICAP_XSCALING

Description

All the X-axis scaling values available. A value of ‘1.0’ is equivalent to 100% scaling. Do not use values less than or equal to zero.

Application

Applications will want to ensure that the values set for this ICAP match those set for ICAP_YSCALING. There are no units inherent with this data as it is normalized to 1.0 being “unscaled.”

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	1.0
Allowed Values:	> 0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_YSCALING

ICAP_YNATIVERESOLUTION

Description

The native optical resolution along the Y-axis of the device being controlled by the Source.

Measured in units of pixels per unit as defined by ICAP_UNITS (pixels per TWUN_PIXELS yields dimensionless data).

Application

Most devices will respond with a single value (TW_ONEVALUE). This is NOT a list of all resolutions that can be generated by the device. Rather, this is the resolution of the device's optics

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	> 0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

None

See Also

ICAP_UNITS
ICAP_XNATIVERESOLUTION
ICAP_YRESOLUTION

ICAP_YRESOLUTION

Description

All the Y-axis resolutions the Source can provide.

Measured in units of pixels per unit as defined by ICAP_UNITS (pixels per TWUN_PIXELS yields dimensionless data). That is, when the units are TWUN_PIXELS, both ICAP_XRESOLUTION and ICAP_YRESOLUTION shall report 1 pixel/pixel. Some data sources like to report the actual number of pixels that the device reports, but that response is more appropriate in ICAP_PHYSICALHEIGHT and ICAP_PHYSICALWIDTH.

Application

Setting this value will restrict the various resolutions that will be available to the user during acquisition.

Applications will want to ensure that the values set for this ICAP match those set for ICAP_XRESOLUTION.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	> 0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT,
MSG_SET/RESET

See Also

ICAP_UNITS
ICAP_XRESOLUTION
ICAP_YNATIVERESOLUTION

ICAP_YSCALING

Description

All the Y-axis scaling values available. A value of ‘1.0’ is equivalent to 100% scaling. Do not use values less than or equal to zero.

There are no units inherent with this data as it is normalized to 1.0 being “unscaled.”

Application

Applications will want to ensure that the values set for this ICAP match those set for ICAP_XSCALING.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

Values

Type:	TW_FIX32
Default Value:	1.0
Allowed Values:	> 0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_XSCALING

ICAP_ZOOMFACTOR

Description

When used with MSG_GET, return all camera supported lens zooming range.

Application

Use this capability with MSG_SET to select one of the lens zooming value that the Source supports.

Source

If not supported, return TWRC_FAILURE / TWCC_CAPUNSUPPORTED.

If Operation is not supported, return TWRC_FAILURE, TWCC_CAPBADOPERATION. (See DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT)

Values

Type:	TW_INT16
Default Value:	0
Allowed Values:	Source dependent.
Container for MSG_GET:	TW_ENUMERATION, TW_ONEVALUE, TW_RANGE
Container for MSG_SET:	TW_ONEVALUE

Required By

None. Highly recommended for digital cameras that are equipped with zoom lenses.

Source Required Operations

MSG_GET, MSG_SET,
MSG_GETCURRENT,
MSG_RESET

10

Return Codes and Condition Codes

Chapter Contents

An Overview of Return Codes and Condition Codes

The TWAIN protocol defines no dynamic messaging system through which the application might determine, in real-time, what is happening in either the Source Manager or a Source. Neither does the protocol implement the native messaging systems built into the operating environments that TWAIN is defined to operate under (Microsoft Windows and Macintosh). This decision was made due to issues regarding platform specificity and higher-than-desired implementation costs.

Instead, for each call the application makes to DSM_Entry( ), whether aimed at the Source Manager or a Source, the Source Manager returns an appropriate Return Code (TWRC_xxxx). The Return Code may have originated from the Source if that is where the original operation was destined.

To get more specific status information, the application can use the DG_CONTROL / DAT_STATUS / MSG_GET operation to inquire the complimentary Condition Code (TWCC_xxxx) from the Source Manager or Source (whichever one originated the Return Code).

The application should always check the Return Code. If the Return Code is TWRC_FAILURE, it should also check the Condition Code. This is especially important during capability negotiation.

There are very few, if any, catastrophic error conditions for the application to worry about. Usually, the application will only have to “recover” from low memory errors caused from allocations in the Source. Most error conditions are handled by the Source Manager or, most typically, by the Source (often involving interaction with the user). If the Source fails in a way that is unrecoverable, it will ask to have its user interface disabled by sending the MSG_CLOSEDSREQ to the application’s event loop.

Currently Defined Return Codes

The following are the currently defined return codes:

TWRC_CANCEL	Abort transfer or the Cancel button was pressed.
TWRC_CHECKSTATUS	Partially successful operation; request further information.
TWRC_DSEVENT	Event (or Windows message) belongs to this Source.
TWRC_ENDOFLIST	No more Sources found after MSG_GETNEXT.
TWRC_FAILURE	Operation failed - get the Condition Code for more information.
TWRC_NOTDSEVENT	Event (or Windows message) does not belong to this Source.
TWRC_SUCCESS	Operation was successful.
TWRC_XFERDONE	All data has been transferred.

Currently Defined Condition Codes

The following are the currently defined condition codes:

TWCC_BADCAP*	Capability not supported by Source or operation (get, set) is not supported on capability, or capability had dependencies on other capabilities and cannot be operated upon at this time (Obsolete, see TWCC_CAPUNSUPPORTED, TWCC_BAPBADOPERATION, and TWCC_CAPSEQERROR).
TWCC_BADDEST	Unknown destination in DSM_Entry.
TWCC_BADPROTOCOL	Unrecognized operation triplet.
TWCC_BADVALUE	Data parameter out of supported range.
TWCC_BUMMER	General failure. Unload Source immediately.
TWCC_CAPUNSUPPORTED*	Capability not supported by Source.
TWCC_CAPBADOPERATION*	Operation (i.e., Get or Set) not supported on capability.
TWCC_CAPSEQERROR*	Capability has dependencies on other capabilities and cannot be operated upon at this time.
TWCC_DENIED	File System operation is denied (file is protected).
TWCC_DOUBLEFEED	Transfer failed because of a feeder error
TWCC_FILEEXISTS	Operation failed because file already exists.
TWCC_FILENOTFOUND	File not found.
TWCC_LOWMEMORY	Not enough memory to complete operation.
TWCC_MAXCONNECTIONS	Source is connected to maximum supported number of applications.
TWCC_NODS	Source Manager unable to find the specified Source.
TWCC_NOTEMPTY	Operation failed because directory is not empty.
TWCC_OPERATIONERROR	Source or Source Manager reported an error to the user and handled the error; no application action required.
TWCC_PAPERJAM	Transfer failed because of a feeder error
TWCC_SEQERROR	Illegal operation for current Source Manager or Source state.
TWCC_SUCCESS	Operation worked.

* TWCC_BADCAP has been replaced with three new condition codes that more clearly specify the reason for a capability operation failure. For backwards compatibility applications should also accept TWCC_BADCAP and treat it as a general capability operation failure. No 1.6 Image Data Sources should return this condition code, but use the new ones instead.

Custom Return and Condition Codes

Although probably not necessary or desirable, it is possible to create custom Return Codes and Condition Codes. Refer to the TWAIN.H file for the value of TWRC_CUSTOMBASE for custom Return Codes and TWCC_CUSTOMBASE for custom Condition Codes. All custom values must be numerically greater than these base values. Remember that the consumer of these custom values will look in your TW_IDENTITY.ProductName field to clarify what the identifier's value means. There is no other protection against overlapping custom definitions.

A

TWAIN Articles

Contents

The articles in this appendix provide additional information about some of the features described in this specification.

Device Events

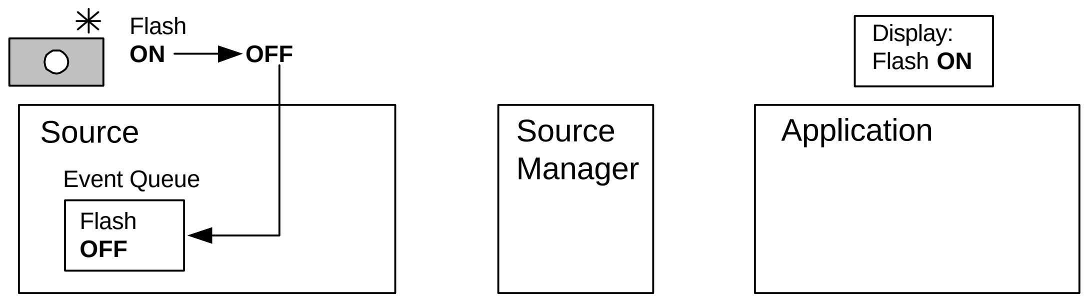
TWAIN 1.8 expands upon asynchronous event notification. Previous versions provided the DG_CONTROL / DAT_NULL messages: MSG_CLOSEDSOK, MSG_CLOSEDSREQ and MSG_XFERREADY to permit the Source to alert the Application that it needed to exit, or that an image was ready to be processed. With the addition of Digital Cameras, and the burgeoning interest in Push Technologies, it has become desirable to enhance TWAIN in this area.

An event begins when the Source needs to alert the Application to some change that has occurred within the device. For example, the owner of a Digital Camera (which is tethered to a host machine) has changed the setting for flash from on to off. The Source wants to alert the Application of this change: first, it records the event in a FIFO queue; second, it sends a DG_CONTROL / DAT_NULL / DAT_DEVICEEVENT to the Source Manager, which forwards the message to the Application.

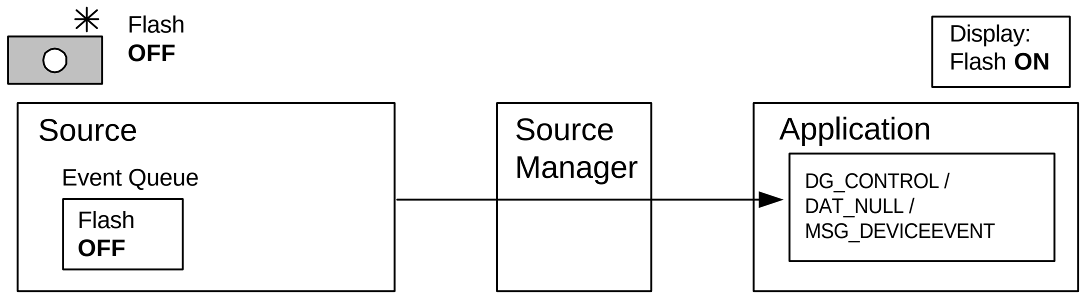
The Application receives the DG_CONTROL / DAT_NULL / DAT_DEVICEEVENT, and immediately issues a DG_CONTROL / DAT_DEVICEEVENT / MSG_GET request to the Source. The Source delivers the information about the event, and pops it off the queue. The process concludes with the Application examining the information and acting upon it, in this case by alerting the user that the flash setting on the camera has been changed.

Notes:

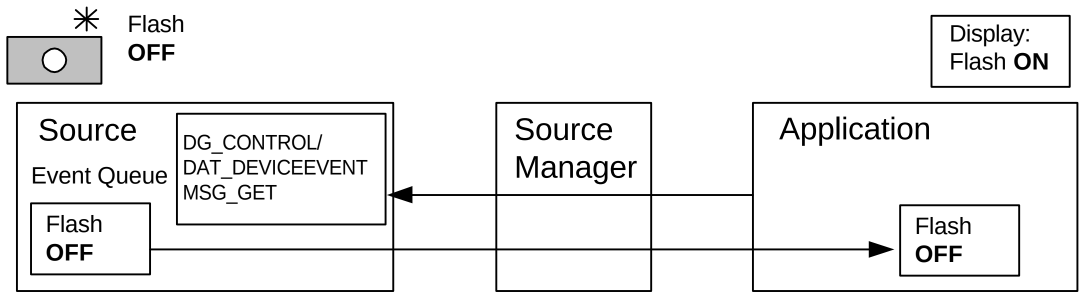
- Sources must start up in a mode with device events turned off (an empty array for CAP_DEVICEEVENTS), this is for the benefit of pre-1.8 applications which may not be able to process this new event.
- Device events are never generated by an Application setting a value within a Source (such as Application changing ICAP_FLASHUSED2). Device events are only generated in response to some outside change within the Source or the Device (such as the User changing the flash setting on the camera).
- Sources must maintain an internal Event Queue, so that they can report each and every device event to the Application in the order of their occurrence.
- Device events are supported in State 4. Windows Sources must use the main window handle supplied with the DG_CONTROL / DAT_PARENT / MSG_OPENDS if they issue device events in State 4. In States 5 through 7 Sources must use the pTW_USERINTERFACE->hParent supplied in the DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS triplet.
- Since device events may occur in State 4, Applications that enable them using CAP_DEVICEEVENTS must be ready to receive and process them.
- When the Application receives a device event, it must immediately collect the information about it. The Application must not issue the DG_CONTROL / DAT_DEVICEEVENT / MSG_GET, except when it has received a DG_CONTROL / DAT_NULL / DAT_DEVICEEVENT message.
- The Application must process events without User intervention, this is to prevent situations where the device event queue builds up because a User is not responding to the system.
- Applications may sometimes fail to respond to a Source's device events. A maximum queue size should be selected so that the Source does not exhaust memory. If the queue fills, the Source must do the following:
  - ✓ Turns off device events (resets CAP_DEVICEEVENT to an empty array).
  - ✓ Refuse to set CAP_DEVICEEVENT until the queue is emptied, return TWCC_SEQERROR.
  - ✓ Process DG_CONTROL / DAT_DEVICEEVENT / MSG_GET requests for each item on the device event queue.
  - ✓ After the last device event is read by the Application, return TWRC_FAILURE / TWCC_DEVICEEVENTOVERFLOW for the next call to DG_CONTROL / DAT_DEVICEEVENT / MSG_GET.
  - ✓ After TWCC_DEVICEEVENTOVERFLOW has been reported, permit the Source to set CAP_DEVICEEVENT again.



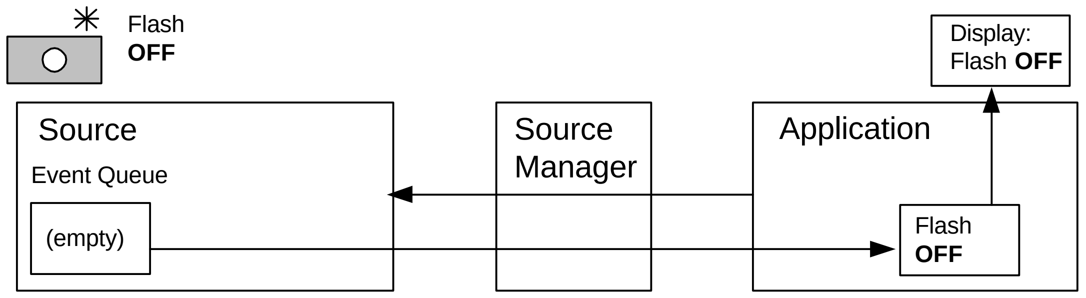
Step 1: The Source senses that the device has changed from ON to OFF and stores this information in an Event Queue. A Queue must be used because the Source may generate multiple events before the Application can respond.



Step 2: The Source sends a DG_CONTROL / DAT_NULL / MSG_DEVICEEVENT to the Application. The Application only knows that some Event has taken place.



Step 3: The Application sends a DG_CONTROL / DAT_DEVICEEVENT / MSG_GET to the Source to learn about the Event. The Source informs the Application that the flash is OFF and it clears the Event from its Queue.



Step 4: The Application informs the User that the flash is now OFF.

Device Events

This section details the various event types and how Sources and Applications should make use of them.

TWDE_CHECKAUTOMATICCAPTURE

The automatic capture settings on the device have been changed.

TWDE_CHECKBATTERY

Status of the battery has changed. Sources will report BatteryMinutes or BatteryPercentage depending on which capabilities say they support.

TWDE_CHECKDEVICEONLINE

The device has been powered off. If an Application receives this device event, it should call CAP_DEVICEONLINE to verify the state of the Source, and then proceed as seems appropriate.

TWDE_CHECKFLASH

The flash setting on the device has been changed.

TWDE_CHECKPOWERSUPPLY

The power supply has changed, for example this event would be generated if AC was removed from a device, putting it on battery. Scanners may also provide this event to notify that a power on reset has taken place, indicating that the device has been power cycled.

TWDE_CHECKRESOLUTION

The resolution on the device has changed.

TWDE_DEVICEADDED

A device has been added to the Source. See DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY and DG_CONTROL / DAT_FILESYSTEM / MSG_GETINFO to get more information about the new device.

TWDE_DEVICEOFFLINE

A device has become unavailable. This is different from TWDC_DEVICEREMOVED, since the device is assumed to be connected.

TWDE_DEVICEREADY

A device is ready to capture another image. Applications should be careful when negotiating this event, especially in situations where images are gathered quickly, as with automatic capture.

TWDE_DEVICEREMOVED

A device has been removed from the Source. This is different from TWDE_DEVICEOFFLINE. As soon as this event is received an Application should re-negotiate its current device, since that may have been the one that was removed. Sources must default to the TWFY_CAMERA device if the current device is removed.

TWDE_PAPERDOUBLEFEED

Report double feeds to the Application. Because of the asynchronous nature of device events there may still be images waiting to be transferred, applications need to decide if they want to recover these images or discard them.

TWDE_PAPERJAM

Report paper jams to the Application. Because of the asynchronous nature of device events there may still be images waiting to be transferred, applications need to decide if they want to recover these images or discard them.

Supported Sizes

Typical uses for ICAP_SUPPORTEDSIZES include, but are not limited to the following:

A0, A1	technical drawings, posters
A2, A3	drawings, diagrams, large tables
A4	letters, magazines, forms, catalogs, laser printer and copying machine output
A5	note pads
A6	postcards
B5, A5, B6, A6	books
C4, C5, C6	envelopes for A4 letters: unfolded (C4), folded once (C5), folded twice (C6)
B4, A3	newspapers, supported by most copying machines in addition to A4

The following table details the physical dimensions associated with ICAP_SUPPORTEDSIZES. Multiply millimeters by 0.03937 to get the approximate inches. Multiply inches by 25.4 to get the approximate millimeters.

ICAP_SUPPORTEDSIZES	**Description**
TWSS_NONE	Images will match the maximum scanning dimensions of the device. This setting is only applicable to devices that have fixed measurable dimensions, such as most scanners. Devices that do not support physical dimensions should not support ICAP_SUPPORTEDSIZES.
TWSS_A4LETTER TWSS_B5LETTER TWSS_B3 TWSS_B4 TWSS_B6	These values are preserved for backward compatibility. TWAIN 1.8+ enabled Applications should not use these settings.
TWSS_B	This value is obsolete, and no longer supported by the specification. Do not use it.
TWSS_USLETTER	8.5” x 11.0” (216mm x 280mm)
TWSS_USLEGAL	8.5” x 14.0” (216mm x 356mm)
TWSS_USLEDGER	11.0” x 17.0” (280mm x 432mm)
TWSS_USEXECUTIVE	7.25” x 10.5” (184mm x 267mm)
TWSS_USSTATEMENT	5.5” x 8.5” (140mm x 216mm)

TWSS_BUSINESSCARD	90mm	x	55mm
TWSS_4A0	1682mm	x	2378mm
TWSS_2A0	1189mm	x	1682mm
TWSS_A0	841mm	x	1189mm
TWSS_A1	594mm	x	841mm
TWSS_A2	420mm	x	594mm
TWSS_A3	297mm	x	420mm
TWSS_A4	210mm	x	297mm
TWSS_A5	148mm	x	210mm
TWSS_A6	105mm	x	148mm
TWSS_A7	74mm	x	105mm
TWSS_A8	52mm	x	74mm
TWSS_A9	37mm	x	52mm
TWSS_A10	26mm	x	37mm
TWSS_ISOB0	1000mm	x	1414mm
TWSS_ISOB1	707mm	x	1000mm
TWSS_ISOB2	500mm	x	707mm
TWSS_ISOB3	353mm	x	500mm
TWSS_ISOB4	250mm	x	353mm
TWSS_ISOB5	176mm	x	250mm
TWSS_ISOB6	125mm	x	176mm
TWSS_ISOB7	88mm	x	125mm
TWSS_ISOB8	62mm	x	88mm
TWSS_ISOB9	44mm	x	62mm
TWSS_ISOB10	31mm	x	44mm
TWSS_JISB0	1030mm	x	1456mm
TWSS_JISB1	728mm	x	1030mm
TWSS_JISB2	515mm	x	728mm
TWSS_JISB3	364mm	x	515mm
TWSS_JISB4	257mm	x	364mm
TWSS_JISB5	182mm	x	257mm
TWSS_JISB6	128mm	x	182mm
TWSS_JISB7	91mm	x	128mm
TWSS_JISB8	64mm	x	91mm
TWSS_JISB9	45mm	x	64mm
TWSS_JISB10	32mm	x	45mm

TWSS_C0	917mm	x	1297mm
TWSS_C1	648mm	x	917mm
TWSS_C2	458mm	x	648mm
TWSS_C3	324mm	x	458mm
TWSS_C4	229mm	x	324mm
TWSS_C5	162mm	x	229mm
TWSS_C6	114mm	x	162mm
TWSS_C7	81mm	x	114mm
TWSS_C8	57mm	x	81mm
TWSS_C9	40mm	x	57mm
TWSS_C10	28mm	x	40mm

Automatic Capture

Automatic image capture is intended for Digital Cameras, although there may be opportunities for other kinds of devices. The intention is to allow an Application to control when pictures are taken, how many pictures are taken, and the interval of time between picture taking. All that is required is that the device be able to perform capture on command from the Source, the timing control and storage of pictures may reside in the Source or the device; the Application does not care.

There are three capabilities needed to control automatic capture:

- CAP_AUTOMATICCAPTURE
- CAP_TIMEBEFOREFIRSTCAPTURE
- CAP_TIMEBETWEENCAPTURES

And one triplet:

- DG_CONTROL / DAT_FILESYSTEM / MSG_AUTOMATICCAPTUREDIRECTORY

CAP_AUTOMATICCAPTURE selects the number of images to be captured. A value of zero (0), the default, disables it. CAP_TIMEBEFOREFIRSTCAPTURE selects how many milliseconds are to pass before the first picture is taken by the device. If this value is 0, then picture taking begins immediately. CAP_TIMEBETWEENCAPTURES selects the milliseconds of elapsed time between pictures. If this value is 0, then the pictures are taken as fast as the device can go.

DG_CONTROL / DAT_FILESYSTEM / MSG_AUTOMATICCAPTUREDIRECTORY selects the directory that will receive the images as they are captured.

Automatic capture expects the device (or Source) to manage the storage of images until the Application is ready to collect them. Applications may choose to retrieve images as they are captured by the Source (using the DAT_FILESYSTEM triplets to browse the storage directory), but must realize that this may affect the performance of the device.

The nature of automatic capture suggests that an Application should be able to disconnect from a Source and expect that if it returns after CAP_TIMEBEFOREFIRSTCAPTURE has passed, there may be images available for it to collect. Because of this Sources should remember their automatic capture settings from session to session, so that a Source starting up does not inadvertently clear them.

Applications need to remember that since the capture of images may occur outside of their control that the settings may be changed directly on the device by the user, resulting in alternations in any of the automatic capture settings. Applications that cannot support this uncertainty should clear the Source’s automatic capture settings prior to shutdown (and after notifying the User).

Camera Preview

Some digital cameras offer a way to preview the intended shot through either a continuous flow of low-resolution frames or streaming video. TWAIN exposes two methods for a Source to present this information to an Application, both in association with the TWFY_CAMERAPREVIEW device.

The TWFY_CAMERAPREVIEW Device

Sources that wish to provide access to their preview camera must do so through DAT_FILESYSTEM. A minimum configuration includes a single TWFY_CAMERA and a single TWFY_CAMERAPREVIEW. The Application discovers what devices are available by using the DAT_FILESYSTEM commands MSG_GETFIRSTFILE and MSG_GETNEXTFILE. It can then switch from the startup default TWFY_CAMERA to the TWFY_CAMERAPREVIEW using the MSG_CHANGEDIRECTORY command.

Performance

It is important when taking a picture from preview mode that the switch from TWFY_CAMERAPREVIEW to TWFY_CAMERA happens as quickly as possible. Applications can minimize the switch over time by negotiating the settings of the TWFY_CAMERA before changing to the TWFY_CAMERAPREVIEW device to collect real-time images.

Sources can help by optimizing their communication with the TWFY_CAMERA, perhaps downloading its values when the user sends MSG_ENABLEDS to the TWFY_CAMERAPREVIEW device so that when the switch back occurs all that needs to happen is a command sent to the camera to take a picture.

Another matter of importance is the transfer mechanism. If the camera is capable of sending a run of continuous snapshots to the application (as opposed to real video streaming), then it is recommended that the TWFY_CAMERAPREVIEW device only support an ICAP_XFERMECH of TWSX_NATIVE.

Entering Preview Mode

An application should do the following before entering preview mode.

1. The application sends MSG_OPENDS to the Source.
2. The application determines that the Source TWFY_CAMERAPREVIEW device.
3. The user / application negotiates values for the TWFY_CAMERA device.
4. The user / application decides to enter preview mode. The application uses MSG_CHANGEDIRECTORY to change to the TWFY_CAMERAPREVIEW device.
5. The application uses MSG_ENABLEDS to enter preview mode. Note that the value of ShowUI should depend on which of the next two sections the application decides to use to control the Source (GUI mode or programmatic).

Previewing with the Source's GUI (ShowUI == TRUE)

If the application relies solely on the Source's GUI for its control of the camera, then it shouldn't have to worry about preview mode issues, since it is hoped that a Source that supports preview will provide access to it from its GUI. This section is concerned with a more limited area, where an application has opted to control the Source programmatically, except for the use of preview. One reason an application might need to do this is to provide preview support for cameras that output streaming video. TWAIN does not have a mechanism for handling this kind of data, so if the only way that a TWAIN application will be able to show this kind of preview data, is if the Source provides a GUI that can show it.

If the Source has CAP_CAMERAPREVIEWUI set to TRUE, then it is possible for the application to use this to preview the images coming from the camera. In this mode the application does not have to concern itself with the kind of data that the Source is providing, since the Source takes the responsibility of displaying the preview images to the user. However, the application does have to wait for the triggers that indicates that the user wishes to take a picture, or that they wish to exit from preview mode. To help standardize this behavior, the preview GUI should be able to indicate two things.

1. **Take a picture** – if the user selects to take a picture, perhaps by pressing a button labeled CAPTURE, then the Source should send the DAT_NULL command MSG_CLOSEDSOK back to the application.
2. **Cancel preview** – if the user decides to exit from preview mode, then the Source should send the DAT_NULL command MSG_CLOSEDSREQ back to the application. The application should then send MSG_DISABLEDS to the Source, change back to the TWFY_CAMERA device, and resume its programmatic control of the Source.

Previewing under Programmatic Control (ShowUI == FALSE):

TWAIN provides programmatic support for TWFY_CAMERAPREVIEW devices that operate by taking a continuous flow of low-resolution snapshots. An application learns that a Source is capable of this by changing to TWFY_CAMERAPREVIEW and testing ICAP_XFERMECH. If the capability is supported, then the TWFY_CAMERAPREVIEW device is capable of transferring these low-resolution images fast enough to simulate real-time video. The way the application obtains these images is similar to how scanners work. The application sets CAP_XFERCOUNT to –1 and enables the Source. The Source sends a MSG_XFERREADY to the application, and the application begins transferring and displaying the low-resolution images as fast as it can. These steps are repeated to aid understanding…

1. The application negotiates any capabilities with the TWFY_CAMERAPREVIEW device, including setting CAP_XFERCOUNT to –1, indicating that the application wishes to receive an unlimited number of images.
2. The application send MSG_ENABLEDS (ShowUI == FALSE) to the Source.
3. The Source sends back MSG_XFERREADY and transitions to State 6.
4. The application uses MSG_IMAGENATIVEXFER to transfer the image and the Source transitions to State 7.
5. The application displays the image.
6. The application uses DAT_PENDINGXFERS / MSG_ENDXFER to transition the Source to State 6. The application needs to pay attention to the TW_PENDINGXFERS.Count, but it is expected that it should remain at –1.
7. Go to step (4).

As long as the application and Source are looping from steps (4) through (7) the application should be displaying a continuous run of snapshots.

Since the application is in complete control, it is implementation dependent on how the user indicates that a picture should be taken. However, once the decision to take a picture is made, the steps to do it are as follows…

Taking a Picture:

The application should do the following when it is told to take a picture while in preview mode.

1. The application sends DAT_PENDINGXFERS / MSG_ENDXFER to the Source, transitioning from State 7 to State 6 (if necessary).
2. The application sends DAT_PENDINGXFERS / MSG_RESET to the Source, transitioning from State 6 to State 5.
3. The application sends MSG_DISABLEDS to the Source, transitioning from State 5 to State 4.
4. The application uses MSG_CHANGEDIRECTORY to switch from the TWFY_CAMERAPREVIEW device to the TWFY_CAMERA device.
5. The application uses MSG_ENABLEDS (ShowUI == FALSE) to enable the TWFY|_CAMERA device.
6. The application sends one of the MSG_IMAGExxxxXFER commands to the Source.
7. The source takes the full resolution picture and transfers it back to the application

File System

This section consists of the following:

- Overview
- Rules for path and file names
- File system components
- Rule for root directory
- Rules for image directory
- File Types
- DAT_FILESYSTEM operations
- Thumbnails and Sound snippets
- Context variable
- Condition Codes

Note: The term ‘camera’ is used generically in the specification to describe a device that captures an image, and is not limited to just devices that employ a camera to accomplish this.

Overview

Digital cameras and some scanners have the ability to capture images to their own local storage. When Automatic Capturing is being used an Application need not collect the captured images until long after their acquisition. A file system is a good representation for the storage of images (since it is a model that is familiar to most programmers), so TWAIN exposes a simple file system interface that Applications may browse through in a random fashion.

There is also a need in TWAIN to expose multiple devices through a single Source. Single pass duplex scanners have multiple cameras that accept different settings. Digital cameras come with disks and memory expansion cards, and many are able to provide a stream of preview images. The file system offers a way for a Source to maintain in its root directory a list of the devices available to an Application.

Rules for path and file names

There are two main grouping of files supported by TWAIN; devices, which are associated with real-time capture, which accept image capture settings, and which are of the form:

/DeviceName

And image path and file names, which are images on local storage which have been previously captured by the device, and which are of the form (bracketed items are optional):

[/DomainName] [/HostName] /TopDirectory [/Sub-Directory…] /ImageFile

1. A filename consists of any characters *except*: NUL (0), either of the slashes '/' or '\' and the colon ':'.
2. Sources should at a minimum support the characters: "A-Z a-z 0-9 _ ."
3. The file system should not be case sensitive, though it may show upper and lowercase.
4. Applications should take into consideration that internationalized Sources may construct filenames from characters within UNICODE.
5. The forward slash '/' and backward slash '\' may be used interchangeably in the creation of path names. Sources and Applications must support the use of both slashes. (ex: /abc\xyz).
6. Multiple adjacent slashes reduce to a single slash. (ex: ///\\abc///xyz == /abc/xyz).
7. The root directory is designated as a solitary slash (ex: / or \).
8. The MSG_CHANGEDIRECTORY and MSG_AUTOMATICCAPTUREDIRECTORY operations are the only ones that accepts absolute or relative directory paths. All other operations occur within the current directory.
9. MSG_CHANGEDIRECTORY and MSG_AUTOMATICCAPTUREDIRECTORY can use dot '.' to address the current directory (ex: ./abc).
10. MSG_CHANGEDIRECTORY and MSG_AUTOMATICCAPTUREDIRECTORY can use dot-dot '..' to address the parent directory (ex: ../abc).
11. In the root directory a MSG_CHANGEDIRECTORY or AUTOMATICCAPTUREDIRECTORY to dot-dot '..' is the same as dot '.' (ex: /. == /..).

Examples:

\Camera is the same as /Camera
//Camera is the same as /Camera
./Camera is the same as /Camera
../Camera is the same as /Camera

File System components

A file system consists of the following.

1. A root directory.
2. A camera device (TWFY_CAMERA), which must be the default device when the Source starts.
3. Zero or more additional devices (TWFY_CAMERATOP, TWFY_CAMERATOP, TWFY_CAMERAPREVIEW).
4. It is possible for a Source to support multiples of a given device type, for instance a scanner may support two devices of type TWFY_CAMERA, both with a supporting TWFY_CAMERATOP and TWFY_CAMERABOTTOM. Use pTW_FILESYSTEM->DeviceGroupMask to uniquely identify a camera or to group it with its associated top and bottom cameras. For example:

Name	Type	Group
/camera_1	TWFY_CAMERA	0x0001
/camera_1_top	TWFY_CAMERATOP	0x0001
/camera_1_bottom	TWFY_CAMERABOTTOM	0x0001
/camera_2	TWFY_CAMERA	0x0002
/camera_2_top	TWFY_CAMERATOP	0x0002
/camera_2_bottom	TWFY_CAMERABOTTOM	0x0002

5. Zero or more directories for storing images (on memory cards, disks, etc…). These are organized in a hierarchical structure that permits, but does not require the ability to browse in a network:

   A TWFY_DOMAIN directory contains only TWFY_HOST directories

   A TWFY_HOST directory contains only TWFY_DIRECTORY directories

   A TWFY_DIRECTORY contains TWFY_IMAGE files and/or TWFY_DIRECTORY directories.

   Sources that provide image storage must provide at least one TWFY_DIRECTORY. TWFY_DOMAIN and TWFY_HOST are optional.

Rules for root directory

1. The root directory can only contain devices or directories, not images.
2. The application cannot create, delete, copy into or rename files in the root directory.
3. Files in a directory are not ordered in any fashion (for instance, an Application may not assume that they are alphabetically sorted). There is one exception to this rule: when an Application issues a DG_CONTROL / DAT_FILESYSTEM / MSG_GETFIRSTFILE on the root directory, the Source must return a TWFY_CAMERA device. This device is the designated default capture camera. If an Application begins capability negotiation, or image capture *without* accessing DAT_FILESYSTEM, then this is the device that will be used.

Rules for image directory

1. A TWFY_DIRECTORY can contain 0 or more TWFY_DIRECTORYs (sub-directories).
2. Can contain 0 or more TWFY_IMAGE (image files).
3. May be fully accessible, read or write protected.
4. May be created or deleted by an Application, given that it is not in the root directory, and that it is not protected by the Source.

Context variable:

The reason for the Context variable is that it allows for unconditional mingling of DAT_FILESYSTEM operations. If there was no Context variable, then Applications would be more limited in the order of operations that could be performed. For instance, the recursive directory walk in the code sample would be much harder to accomplish without a Context to help the Source identify the current directory being accessed by a call to MSG_GETNEXTFILE.

This value is provided solely for the benefit of Source writers. When MSG_GETFIRSTFILE is called, the Source should record the current directory and the current file and store those values internally, using Context as a reference to their location. The nature or value of the Context is dependent on the implementation of the Source, Applications must never attempt to use or modify the Context. A call to MSG_GETINFO must use this Context to identify the file being reported. Calls to any of the file transfer methods (MSG_IMAGENATIVEXFER, MSG_IMAGEFILEXFER, MSG_IMAGEMEMXFER, MSG_AUDIONATIVEXFER, MSG_AUDIOFILEXFER) must use this Context to determine the data being sent to the Application. A call to MSG_GETNEXTFILE must use this Context to help obtain the next file from the directory (this will result in a change in the context as it references the new file). And, finally, a call to MSG_GETCLOSE releases the memory in the Source associated with this Context.

Condition Codes:

These are some condition codes that apply specifically to file system operations:

TWCC_DENIED	File system operation is denied. A Source should report this condition code if an attempt is made to access a protected file. Examples of such protection include: any attempt to delete, rename or copy into the root directory; protected files that are on the network; and any file that the Source feels it needs to protect.
TWCC_FILEEXISTS	The operation failed because the file already exists. A Source should report this condition code if an attempt is made to create a sub-directory with a name that already exists in the targeted directory; or if an attempt is made to copy or rename over an existing file or directory.
TWCC_FILENOTFOUND	The file was not found. This can occur for a variety of reasons: attempts to change directory to a path that does not exist; attempts to delete, rename or copy files that do not exist; as the condition code from MSG_GETFIRSTFILE for an empty directory; or MSG_GETNEXTFILE when it finds no more files in the current directory; and, finally, from MSG_GETINFO if it is requested to provide information on a file that has been deleted.
TWCC_NOTEMPTY	Operation failed because the directory is not empty. This condition code is used by the Source if an attempt is made with the Recursion flag set to FALSE to delete a non-empty directory.

File Types

The DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY operation is used to make either a device or a directory current. If a camera device is the target, then all capability negotiation is with that device and all images come from that device, until a new MSG_CHANGEDIRECTORY command is issued. If an image directory is selected then the current device is set to be the root level directory name (i.e., changing to /abc/mno/xyz means that the current device is /abc).

TWFY_CAMERA	Every TWAIN file system must support at least one camera, which must be the default device on startup. This is for compatibility with pre 1.8 applications as well as post 1.8 applications that do not choose to make use of the file system. On single pass duplex scanners, this camera device is used to simultaneously set values for the top and bottom cameras. During the capturing of images (in duplex mode) it sends a stream of images in the order: TOP, BOTTOM, TOP…
TWFY_CAMERATOP / TWFY_CAMERABOTTOM	Single pass duplex scanners may opt to provide independent access to the top and bottom cameras. A device with one of these file types controls the settings for the specified camera. If this device is the current device at the time image capture commences, then only images from that camera will be passed to the Application. This means that even if a device is set for duplex scanning, if the current device has a file type of TWFY_CAMERATOP, then only top images will be passed to the Application.
TWFY_CAMERAPREVIEW	A logical device that performs camera live preview functionality. When implementing the Source for this logical device, related capabilities must be negotiated to perform preview specific functions. Among them, ICAP_XRESOLUTION and ICAP_YRESOLUTION must be implemented to specify the preview image sizes. Other capabilities may be available in some sources, such as ICAP_ZOOMFACTOR and ICAP_FLASHUSED2.

TWFY_DIRECTORY	At the root directory level files of this type should correspond to a physical piece of hardware (a memory card or a disk). The root directory is only allowed to contain devices. Sub-directories may only contain image files or more sub-directories. Access to files and directories is controlled by the Source, so Applications should check all operations and watch out for condition codes such as TWCC_DENIED.
TWFY_IMAGE	Any directory, except root, may contain image files. The DAT_FILESYSTEM messages MSG_GETFIRSTFILE and MSG_GETNEXTFILE select the current image. Once an image has been selected, it may be transferred in the same fashion used to acquire images from a camera. Note: this file type is reserved for full resolution images, see the section on Thumbnails for information on how to acquire them.

DAT_FILESYSTEM Operations

MSG_AUTOMATICCAPTUREDIRECTORY	
	Selects the directory to be used to store images acquired by automatic capture.
MSG_CHANGEDIRECTORY	
	Selects the device or image subdirectory. Use this to select between direct camera (scanner) control, and browsing of stored images. All capabilities negotiated and triplet operations are with the current device (directory), until this value is changed by the Application.
MSG_COPY	Copies the specified file from one directory to another. If the Recursive flag is TRUE and the file type specified is TWFY_DIRECTORY then that directory and all the files and directories under it are copied. The Application cannot copy files into the root directory.
MSG_CREATEDIRECTORY	
	Creates a new image subdirectory. The Application cannot create files in the root directory.
MSG_DELETE	Deletes the specified file. If the Recursive flag is TRUE and the file type specified is TWFY_DIRECTORY, then all the files under that directory are deleted. The Application cannot delete files in the root directory.
MSG_FORMATMEDIA	Formats the currently selected storage device. Use with caution.

MSG_GETCLOSE	Closes the Context created by MSG_GETFIRSTFILE.
MSG_GETFIRSTFILE	Creates a Context that points to the first file in a directory. This Context is used by MSG_GETINFO, MSG_GETNEXTFILE, MSG_GETCLOSE; and for files of type TWFY_IMAGE all image transfer related operations performed in states 6 and 7 use the image pointed to by this Context (i.e., DAT_IMAGEINFO, DAT_IMAGEMEMXFER, etc…).
MSG_GETINFO	Returns information about a device, directory or image file.
MSG_GETNEXTFILE	Updates the Context to point to the next file in the directory.
MSG_RENAME	Renames a directory or an image file. If the directories differ, then it moves the file as well, creating it in the new location and deleting it from the old location. Files in the root directory cannot be renamed by the Application.

Thumbnails and Sound snippets

TWAIN is primary concerned with the acquisition of images, so the file system does not contain thumbnail files or sound files, since these kinds of data are expected to be associated with image files. This simplifies an Application's browsing of the file system, since it need only concern itself with one type of data file (TWFY_IMAGE), and does not have to trace associated data files.

Sources must filter out non-image files, if the device stores thumbnail and sound data independent of the image files. For instance, if a device stores the following files:

IMAGE001.TIF
IMAGE001_THUMBNAIL.TIF
IMAGE001_SOUND.WAV

The file system must only report the existence of IMAGE001.TIF

An Application obtains the thumbnail for an image by setting ICAP_THUMBNAILSENABLED to TRUE; the same filename is used for both the full resolution and thumbnail versions of an image. By setting ICAP_THUMBNAILSENABLED, the Application decides which version of the image it receives.

Sound snippets are also associated with image files, unlike thumbnails it is possible for a single image file to own several sound snippets. An Application can get the number of snippets that an image owns, and then, during image transfer, the Application has the option to transfer any number of those snippets. It is also possible to collect the snippets for an image without transferring the image data.

Sample Recursive Directory Walk

The following is a sample recursive directory walk.

```
// This Application function walks through all the files in a Source’s
// file system, counting the file types file system, counting the file
// types it finds.  It is intended only as a sample, error checking is
// omitted to simplify the code.
typedef struct {
   int Devices;
   int Directories;
   int Images;
} t_Counters;
TW_UINT16 DirectoryWalk(TW_FILESYSTEM *fsArg, t_Counters *Counters)
{
   TW_UINT16 rc;  TW_FILESYSTEM fs;
   // Caller has set fsArg->InputFile to some value, such as “/”…
   rc = (*DS_Entry) (&app,&src,DG_CONTROL,DAT_FILESYSTEM,
      MSG_CHANGEDIRECTORY, fsArg);

   // We do GETFIRSTFILE first in each new directory, GETNEXTFILE for all
   // subsequent calls…
   for (rc = (*DS_Entry)(&app,&src,DG_CONTROL,DAT_FILESYSTEM,
      MSG_GETFIRSTFILE,&fs);
       rc == TWRC_SUCCESS;
      rc = (*DS_Entry)(&app,&src,DG_CONTROL,DAT_FILESYSTEM
         ,MSG_GETNEXTFILE,&fs)) {

         // Count the appropriate file type…
         switch (fs.FileType) {
             default:  Counters->Devices += 1;  break;
             case TWFY_IMAGE:  Counters->Images  += 1;  break;
             case TWFY_DOMAIN:
             case TWFY_HOST:
             case TWFY_DIRECTORY:
                  Counters->Directories += 1;
                 // Recursively step into this directory, looking for
more
                 // stuff…
                  rc = DirectoryWalk(&fs,&Counters);
                  if (rc != TWRC_SUCCESS) {
                     rc =
(*DS_Entry)(&app,&src,DG_CONTROL,DAT_FILESYSTEM,
                       MSG_GETCLOSE,&fs);
                     return(rc);
                  }
                  break;
         }
   }
   // Cleanup and return…
   rc =
(*DS_Entry)(&app,&src,DG_CONTROL,DAT_FILESYSTEM,MSG_GETCLOSE,&fs);
   return(TWRC_SUCCESS);
}

// Using this function…
TW_UINT16 rc;
TW_FILESYTEM fs;
t_Counters Counters;
memset(&fs,0,sizeof(fs));
memset(&Counters,0,sizeof(Counters));
strcpy(fs.InputFile,"/"); // start at root…
rc = DirectoryWalk(&fs,&Counters);
```

Internationalization

A TWAIN Source can easily be internationalized despite its 8-bit character interface. A well designed Source should automatically match the locale of the application calling it; passing localized data through the API, and displaying appropriate language text in its user interface. Developers have the option of using UNICODE or MultiByte encodings, the 8-bit interface is not an obstacle to Applications or Sources.

When an Application calls DG_CONTROL / DAT_IDENTITY / MSG_OPENDS, it provides to the Source its TW_IDENTITY data. Internationalized Sources should check the appIdentity->Version.Language field, and attempt to match the Application's language (returning the same value in the dsIdentity structure). If the Source is incapable of matching the language, then it should attempt to match the User's current locale (on Win32 do this using the LOCALE_USER_DEFAULT value returned by the GetLocaleInfo() call). In most cases the Application locale and the User locale will be the same, and the Source will have to select the best language it can. For instance, if the Application requested Swiss French, and the Source only has French, then it should offer that. Otherwise, it should resort to some common secondary language, such as English.

Please note that DG_CONTROL / DAT_IDENTITY / MSG_OPENDS is the very first opportunity that an Application and Source have to negotiate language. DG_CONTROL / DAT_IDENTITY / MSG_GET, when invoked in state 3, does not provide an appIdentity. Sources should default to the LOCALE_USER_DEFAULT in this instance.

As mentioned above, the TWAIN interface assumes 8-bit characters, this prevents the direct passing of UNICODE data between Sources and Applications, but it does not hinder indirect means that convert data into MultiByte encodings. The remainder of this section shows one way of allowing Sources and Applications to communicate, without worrying about whether they are UNICODE or MultiByte enabled. The best example to illustrate this is to consider a Source and Application, both UNICODE enabled, communicating through the TWAIN interface.

To pass UNICODE string data from the Source to the Application, the Source must convert UNICODE to MultiByte, using the appropriate Code-Page (which is specific to a given set of locales). When the Application receives the data, it converts from MultiByte back to UNICODE. The process is the same when sending string data from the Application to the Source. The process depends on the Application and Source using the same Code-Page for their conversion. The Win32 functions required to perform the conversions are WideCharToMultiByte and MultiByteToWideChar. The only limitation to watch out for is the size of the various strings provided by TWAIN. At all times the MultiByte data must fit within the strings described by the interface, and Source and Application writers need to pay close attention to it.

```
int WideCharToMultiByte(
UINT CodePage,                    // code page
DWORD dwFlags,                    // performance and mapping flags
LPCWSTR lpWideCharStr,            // address of wide-character string
int cchWideChar,                  // number of characters in string
LPSTR lpMultiByteStr,             // address of buffer for new string
int cchMultiByte,                 // size of buffer
LPCSTR lpDefaultChar,             // address of default for unmappable characters
LPBOOL lpUsedDefaultChar          // address of flag set when default char. used
);
int MultiByteToWideChar(
UINT CodePage,                    // code page
DWORD dwFlags,                    // character-type options
LPCSTR lpMultiByteStr,            // address of string to map
int cchMultiByte,                 // number of characters in string
LPWSTR lpWideCharStr,             // address of wide-character buffer
int cchWideChar                   // size of buffer
);
```

These functions are fully described in the online Microsoft Visual C++ documentation. This section does not attempt to duplicate that information, but does show how Source and Application may cooperate when using them to transmit localized data through the TWAIN interface.

TWAIN CAP_LANGUAGE Code to ANSI Code-Page Table

```
// This array maps TWAIN CAP_LANGUAGE codes to the appropriate ANSI Code-
// Page.  There is no mechanism for converting to the OEM Code-Page, nor
// should one be needed, since the upper 128 bytes in the OEM pages
mostly
// contain line art characters used by MS-DOS.
// Note:  the index in the comment field is just an index into the array,
// it does not correspond to the TWAIN constant for a given TWLG field…
//

#define AnsiCodePageElements 88
int AnsiCodePage[AnsiCodePageElements] = {
   1252,       //   0     TWLG_DANISH            (TWLG_DAN)
   1252,       //   1     TWLG_DUTCH           (TWLG_DUT)
   1252,       //   2     TWLG_ENGLISH         (TWLG_ENG)
   1252,       //   3     TWLG_FRENCH_CANADIAN (TWLG_FCF)
   1252,       //   4     TWLG_FINNISH         (TWLG_FIN)
   1252,       //   5     TWLG_FRENCH          (TWLG_FRN)
   1252,       //   6     TWLG_GERMAN          (TWLG_GER)
   1252,       //   7     TWLG_ICELANDIC       (TWLG_ICE)
   1252,       //   8     TWLG_ITALIAN         (TWLG_ITN)
   1252,       //   9     TWLG_NORWEGIAN       (TWLG_NOR)
   1250,       //  10     TWLG_PORTUGUESE      (TWLG_POR)
   1252,       //  11     TWLG_SPANISH         (TWLG_SPA)
   1252,       //  12     TWLG_SWEDISH         (TWLG_SWE)
   1252,       //  13     TWLG_ENGLISH_USA     (TWLG_USA)
   1252,       //  14     TWLG_AFRIKAANS
   1250,       //  15     TWLG_ALBANIA
   1256,       //  16     TWLG_ARABIC
   1256,       //  17     TWLG_ARABIC_ALGERIA
   1256,       //  18     TWLG_ARABIC_BAHRAIN
   1256,       //  19     TWLG_ARABIC_EGYPT
   1256,       //  20     TWLG_ARABIC_IRAQ
   1256,       //  21     TWLG_ARABIC_JORDAN
   1256,       //  22     TWLG_ARABIC_KUWAIT
   1256,       //  23     TWLG_ARABIC_LEBANON
   1256,       //  24     TWLG_ARABIC_LIBYA
   1256,       //  25     TWLG_ARABIC_MOROCCO
   1256,       //  26     TWLG_ARABIC_OMAN
   1256,       //  27     TWLG_ARABIC_QATAR
   1256,       //  28     TWLG_ARABIC_SAUDIARABIA
   1256,       //  29     TWLG_ARABIC_SYRIA
   1256,       //  30     TWLG_ARABIC_TUNISIA
   1256,       //  31     TWLG_ARABIC_UAE   /* United Arabic Emirates */
   1256,       //  32     TWLG_ARABIC_YEMEN
   1252,       //  33     TWLG_BASQUE
   1251,       //  34     TWLG_BYELORUSSIAN
   1251,       //  35     TWLG_BULGARIAN
   1252,       //  36     TWLG_CATALAN
    936,       //  37     TWLG_CHINESE
    950,       //  38     TWLG_CHINESE_HONGKONG
    936,       //  39     TWLG_CHINESE_PRC /* People's Republic of China
*/
    936,       //  40     TWLG_CHINESE_SINGAPORE
    936,       //  41     TWLG_CHINESE_SIMPLIFIED
    950,       //  42     TWLG_CHINESE_TAIWAN
```

```
 950,       //  43     TWLG_CHINESE_TRADITIONAL
1250,       //  44     TWLG_CROATIA
1250,       //  45     TWLG_CZECH
1252,       //  46     TWLG_DUTCH_BELGIAN
1252,       //  47     TWLG_ENGLISH_AUSTRALIAN
1252,       //  48     TWLG_ENGLISH_CANADIAN
1252,       //  49     TWLG_ENGLISH_IRELAND
1252,       //  50     TWLG_ENGLISH_NEWZEALAND
1252,       //  51     TWLG_ENGLISH_SOUTHAFRICA
1252,       //  52     TWLG_ENGLISH_UK
1257,       //  53     TWLG_ESTONIAN
1250,       //  54     TWLG_FAEROESE
1256,       //  55     TWLG_FARSI
1252,       //  56     TWLG_FRENCH_BELGIAN
1252,       //  57     TWLG_FRENCH_LUXEMBOURG
1252,       //  58     TWLG_FRENCH_SWISS
1252,       //  59     TWLG_GERMAN_AUSTRIAN
1252,       //  60     TWLG_GERMAN_LUXEMBOURG
1252,       //  61     TWLG_GERMAN_LIECHTENSTEIN
1252,       //  62     TWLG_GERMAN_SWISS
1253,       //  63     TWLG_GREEK
1255,       //  64     TWLG_HEBREW
1250,       //  65     TWLG_HUNGARIAN
1252,       //  66     TWLG_INDONESIAN
1252,       //  67     TWLG_ITALIAN_SWISS
 932,       //  68     TWLG_JAPANESE
 949,       //  69     TWLG_KOREAN
1361,       //  70     TWLG_KOREAN_JOHAB
1257,       //  71     TWLG_LATVIAN
1257,       //  72     TWLG_LITHUANIAN
1252,       //  73     TWLG_NORWEGIAN_BOKMAL
1252,       //  74     TWLG_NORWEGIAN_NYNORSK
1250,       //  75     TWLG_POLISH
1252,       //  76     TWLG_PORTUGUESE_BRAZIL
1250,       //  77     TWLG_ROMANIAN
1251,       //  78     TWLG_RUSSIAN
1250,       //  79     TWLG_SERBIAN_LATIN
1250,       //  80     TWLG_SLOVAK
1250,       //  81     TWLG_SLOVENIAN
1252,       //  82     TWLG_SPANISH_MEXICAN
1252,       //  83     TWLG_SPANISH_MODERN
 874,       //  84     TWLG_THAI
1254,       //  85     TWLG_TURKISH
1251,       //  86     TWLG_UKRANIAN
};
```

Sample Converting from WideChar to MultiByte

The following is a sample of converting from WideChar to MultiByte.

```
// This function converts _TCHAR* strings to MultiByte, using the
// appropriate code page.  If the build is ANSI or MBCS, then no
// conversion is needed, the _tcsncpy() function is used.
// If the build is UNICODE, then the Code-Page is determined, and used to
// convert the string to MultiByte using the WideCharToMultiByte()
// function…
//
int CopyTCharToMultibyte
    (char *dst,
    const int sizeof_dst,
    const _TCHAR *src,
    const int twain_language_code)
{
#ifndef _UNICODE
    // MultiByte string copy…
    _tcsncpy(dst,src,sizeof_dst);
    dst[sizeof_dst-1] = 0;
    return(strlen(dst));
#else
    int cp;
    int len;
    _TCHAR cp_str[16];
    if (twain_language_code >= AnsiCodePageElements) {
          // Whoops, don't have one of those…
          return(-1);
    } else if (twain_language_code >= 0) {
          // Lookup the code page…
          cp = AnsiCodePage[twain_language_code];
    } else {
          // Get the User's code page…
          GetLocaleInfo
             (LOCALE_USER_DEFAULT,
             LOCALE_IDEFAULTANSICODEPAGE,
             cp_str,
             sizeof(cp_str));
          cp = _ttoi(cp_str);
    }
    if (IsValidCodePage(cp) == 0) {
          // That code page isn't installed on this system…
          return(-1);
    }
    len = WideCharToMultiByte(
      cp,          // code page
      0,           // performance and mapping flags
      src,         // address of wide-character string
      -1,          // number of characters in string
      dst,         // address of buffer for new string
      sizeof_dst,  // size of buffer (in characters)
      NULL,        // address of default for unmappable characters
      NULL         // address of flag set when default char. used
    );

#endif
}
```

Sample Converting from MultiByte to WideChar

The following is a sample of converting from MuliByte to WideChar.

```
// This function converts multibyte strings to _TCHAR* strings, using
// the appropriate code page.
// If the build is ANSI or MBCS, then no conversion is needed, the
// _tcsncpy() function is used.  If the build is UNICODE, then the
// Code-Page is determined, and used to convert the string to
// _TCHAR* using the MultiByteToWideChar() function…
//

int CopyMultibyteToTChar
    (_TCHAR *dst,
    const int sizeof_dst,
    const char *src,
    const int twain_language_code)
{
#ifndef _UNICODE
    // MultiByte string copy…
    _tcsncpy(dst,src,sizeof_dst);
    dst[sizeof_dst-1] = 0;
    return(strlen(dst));
#else
    int cp;
    int len;
    _TCHAR cp_str[16];
    if (twain_language_code >= AnsiCodePageElements) {
          // Whoops, don’t have one of those…
          return(-1);
    } else if (twain_language_code >= 0) {
          // Lookup the code page…
          cp = AnsiCodePage[twain_language_code];
    } else {
          // Get the User’s code page…
          GetLocaleInfo
            (LOCALE_USER_DEFAULT,
            LOCALE_IDEFAULTANSICODEPAGE,
            cp_str,
            sizeof(cp_str));
          cp = _ttoi(cp_str);
    }
    if (IsValidCodePage(cp) == 0) {
```

```
                // That code page isn't installed on this system…
                return(-1);
          }
          len = MultiByteToWideChar(
            cp,                       // code page
            0,                        // performance and mapping flags
            src,                      // address of wide-character string
            -1,                       // number of characters in string
            dst,                      // address of buffer for new string
            sizeof_dst/sizeof(_TCHAR)     // size of buffer (in characters)
          );
          return(len);
    #endif
    }
```

Sample Use of the Conversion Functions

The following are examples of UNICODE application and UNICODE source.

UNICODE Application

```
int              sts;
int              twain_language_code;
_TCHAR           Author[128];
pTW_ONEVALUE     pvalOneValue;
. . .
// the Application has queried the Source as to what languages it
supports
//and selected TWLG_JAPANESE, storing it in twain_language_code…
. . .
// CAP_AUTHOR is queried, and a value is received…
. . .
// Convert CAP_AUTHOR string to UNICODE…
sts = CopyMultiByteToTChar
         (Author,
         sizeof(Author),
         (char*)&pvalOneValue->Item,
         twain_language_code)
if (sts < 0) {
        // Error…
. . .
}
```

UNICODE Source

```
. . .
int             sts;
int             source_language_code;
_TCHAR          SourceAuthor[128];
pTW_ONEVALUE    pvalOneValue;
. . .
// the Source has been told to use TWLG_JAPANESE, it stores this value
// in source_language_code …
. . .
// CAP_AUTHOR is queried by the Application…
// The Source keeps the value in SourceAuthor…
. . .
// Convert CAP_AUTHOR string to multibyte…
         sts = CopyTCharToMultibyte
         ((char*)&pvalOneValue->Item,
         sizeof(TW_STR128),
         SourceAuthor,
         source_language_code)
if (sts < 0) {
         // Error…
         . . .
}
. . .
// The Source returns the value to the Application…
```

Audio Snippets

Digital Cameras have the ability to acquire audio snippets along with an image. To support this TWAIN 1.8 provides a new data group, DG_AUDIO. Because TWAIN is image-centric, DG_AUDIO operations are dependent on an image context, audio snippets must be associated with an image. When a Source enters into state 6, the Application can opt to transfer any and all audio snippets. The steps required to obtain audio snippets deliberately parallel the steps required to transfer images, to reduce the effort to learn how to access this new kind of data.

The following Data Argument Types (DATs) are supported by DG_AUDIO:

DAT_AUDIOFILEXFER	transfer audio in file format
DAT_AUDIOINFO	info about an audio snippet
DAT_AUDIONATIVEXFER	transfer audio in native format

The following DG_CONTROL (DATs) are supported when DAT_XFERGROUP is set to DG_AUDIO, DATs not mentioned in this list must return TWRC_FAILURE / TWCC_BADPROTOCOL:

DAT_CAPABILITY	no changes to its operation
DAT_EVENT	no changes to its operation
DAT_IDENTITY	no changes to its operation
DAT_NULL	no changes to its operation
DAT_PASSTHRU	no changes to its operation
DAT_PENDINGXFERS	reports number of snippets remaining to be transferred, MSG_ENDXFER and MSG_RESET do not cause the Source to drop to State 5.
DAT_SETUPFILEXFER	selects the audio file format
DAT_STATUS	no changes to its operation
DAT_USERINTERFACE	no changes to its operation
DAT_XFERGROUP	MSG_SET, MSG_GETDEFAULT and MSG_GETCURRENT added to allow switching between data groups. The default value for MSG_GETDEFAULT must be DG_IMAGE. And when the Source starts up, MSG_GETCURRENT must report DG_IMAGE as the current data group, to maintain compatibility with pre-TWAIN 1.8 Applications.

The following capabilities support audio; all capabilities are negotiable at all times (at least in state 4), independent of the current setting of DAT_XFERGROUP:

ACAP_AUDIOFILEFORMAT	negotiate available audio file formats
ACAP_XFERMECH	negotiate audio snippet transfer mechanism

Collecting Audio Snippets

The transfer of an audio snippet was designed to be used when an Application is browsing through a selection of stored images. There is nothing to prevent the transfer of audio when an image is captured in real-time, though TWAIN does require that any audio snippets be transferred before the image is transferred.

A typical transfer may occur in the following way: An Application is browsing through storage managed by the TWAIN Source using MSG_GETFILEFIRST / MSG_GETFILENEXT (see DAT_FILESYSTEM), and finds an image that it wants to work with. The Application enters state 6 by calling DG_CONTROL / DAT_IDENTITY / MSG_ENABLEDS. If the Application wants to find out if there are any audio snippets associated with the image, it can call DG_AUDIO / DAT_AUDIOINFO / MSG_GET. In this example it finds in the TW_AUDIOINFO structure that this image file has three audio snippets associated with it. The Application wants the second audio snippet, so it calls DG_CONTROL / DAT_XFERGROUP / MSG_SET and sets the data group to DG_AUDIO. This call changes the context of the Source, it is now set up to transfer audio data. One effect of this is that a call to DG__CONTROL / DAT_PENDINGXFERS / MSG_GET will report the number of audio snippets (for this image) that remain to be transferred. Because the Application wants the second audio snippet, it must discard the first one, and does this by making a call to DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER. The snippet that it wants is now available to be transferred, and it does this with a call to DG_AUDIO /

DAT_AUDIONATIVEXFER / MSG_GET. The Source moves up into state 7. The Application ends the transfer with a call to DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER.

Because the Application only wanted the second audio snippet, it can return to DG_IMAGE by making a call to DG_CONTROL / DAT_XFERGROUP / MSG_SET. Once this is done, all other commands work in a traditional TWAIN fashion. The Application can opt to transfer or discard the image, even though it did not transfer all of the audio snippets.

There is one more thing to note, if the Application had read the third audio snippet, or if it had issued the DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET command while in DG_AUDIO, the state of the Source would remain at state 6. TWAIN works this way because it is image-centric, the only way to transition from state 6 to state 5 is when it is determined that there are no more images to transfer.

Notes

1. TWAIN 1.8 supports native and file transfers of audio snippets. Buffered mode transfers are not supported, because TWAIN does not have the necessary infrastructure to describe audio data, and it was decided that adding that structure in this release would be overly complex, and probably incomplete.
2. As a general rule, even though many operations are possible with DAT_XFERGROUP set to DG_AUDIO, Applications are encouraged to only change to DG_AUDIO for the length of time it takes to collect an audio snippet, and to stay in DG_IMAGE mode at all other times.
3. Though TWAIN is image-centric, it is possible to envision a TWAIN Source that is only capable of supporting DG_AUDIO. The TWAIN Working Group feels that any such notion is a bad idea, and encourages anyone thinking of doing this to pick on some other API.

How to use the Preview Device

Application switch to the preview logical device

1. The application first tries to switch to the preview logical device using the DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY triplet with TWFY_CAMERAPREVIEW set in InputName field of TW_FILENAME structure. If the returned value is TW_SUCCESS, the application can proceed.
2. After the application successfully switches to the preview device, all subsequent capability negotiation is with the preview device.
3. The application queries the Source with capability CAP_CAMERAPREVIEWUI. If it returns SUCCESS, then the Source is able to assume the responsibility of displaying preview images. The application can choose to use the Source's UI or not when it issues the MSG_ENABLEDS. If the application uses the Source's UI, it will do nothing but wait to issue MSG_DISABLEDS, or wait for a MSG_CLOSEDSREQ from the Source to stop the preview mode. If the application does not use the Source's UI or the Source does not provide a UI, then the application should follow the following steps.

Setting up environments for Preview Mode

4. The application starts negotiation on the Preview size using the ICAP_XRESOLUTION and ICAP_YRESOLUTION capabilities with MSG_GET first. With the returned supported sizes from the Source, the application can set the selected preview sizes using the ICAP_XRESOLUTION and ICAP_YRESOLUTION capabilities with MSG_SET. These two capabilities should be linked through ICAP_XYRESOLUTIONLINKED.
5. Optionally, the application can negotiate the zoom lens value, camera flash state during previewing, etc, with available capabilities such as ICAP_ZOOMFACTOR, ICAP_FLASHUSED2. If application queries for capabilities that are not related to preview device, Source will return TWRC_FAILURE.

Start getting and displaying Preview Thumbnails

6. The application can use the automatic capture feature with CAP_XFERCOUNT to -1 (Application is willing to transfer multiple images).
7. Application issues MSG_ENABLEDS to move to state 5. Upon receiving this message, the Source should start capturing images[1].
8. Source issues MSG_XFERREADY, indicating that an image is present, and state moves to 6.

[1] The Source takes a picture as soon as it receives MSG_ENABLEDS and each time it receives MSG_ENDXFER

LOOP:

9. Application issues DAT_IMAGENATIVEXFER to get image and goes to state 7.

10. Application issues MSG_ENDXFER to return to state 6, and it displays the image. Then if it wants the next preview image, examines pTW_PENDINGXFERS->Count to verify that there is another image, and it goes to LOOP. Source, upon receiving the MSG_ENDXFER message, takes the next picture and returns -1 in the pTW_PENDINGXFERS->Count.

END LOOP

11. If the application wants to end preview mode, it issues DAT_PENDINGXFERS / MSG_RESET. This forces the Source to go to state 5 (CAP_XFERCOUNT is set to 0). If the Source is unable to deliver preview images, it sets pTW_PENDINGXFERS->Count to 0 in reply to the application's MSG_ENDXFER command, and returns to state 5.

12. The application can then issue MSG_DISABLEDS, which returns it to state 4, and now the application can use DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY to change directory to the camera device to take a full resolution picture.

How to take a snapshot from preview scene

1. The application could provide a button or menu item for the user to take a snapshot from the preview scene, for example, a "Take Picture" button. In response to this, the application should use the triplet DG_CONTROL / DAT_FILESYSTEM / MSG_CHANGEDIRECTORY with TWFS_CAMERA set in the TW_FILENAME structure to stop the preview mode.

2. Subsequently, the application can use the automatic capture feature with CAP_XFERCOUNT to 1, CAP_TIMEBEFOREFIRSTCAPTURE to 0 and CAP_AUTOMATICCAPTURE set to 1 to initiate the capture of preview snapshot.

3. When the Source receives the CAP_AUTOMATICCAPTURE, it should capture the preview snapshot, and inform the application with MSG_XFERREADY when it is ready to transfer.

4. After receiving the MSG_XFERREADY, the application should use one of the three standard image transfer methods to transfer the captured image from the Source to the application.

5. At the end of this operation, the application has the option of going back to the preview thumbnail loop.

Imprinter / Endorser

Scanners intended for document imaging sometimes include accessories that let the scanner print data on the documents as it scans them. TWAIN provides basic functionality to negotiate capabilities for imprinter / endorser devices. An imprinter is a general term for any document-printing device. An endorser is more specialized, and is primarily intended as proof of scanning. In addition to the type of printing device, TWAIN offers ways to locate the printer on the scanning path: top or bottom of the sheet of paper, before or after the paper has been scanned. It is the responsibility of the Source to provide the available combinations to the Application. It is the responsibility of the Application to enable the printers that it wants to use, and to establish seed values prior to scanning.

This is a context sensitive scheme, Applications use CAP_PRINTER to discover what printers are available to the Source, and to select each of those printers for negotiation.

CAP_PRINTERENABLED determines whether or not a given printer will be used when scanning begins; a value of TRUE indicates that it will be used, a value of FALSE that it will not be used. Applications must enable a printer before negotiating the seed values.

CAP_INDEX describes an index that counts by ones for every image seen by a given printer.

CAP_PRINTERMODE selects one of three options: print one line of text from CAP_PRINTERSTRING, or multiple lines from CAP_PRINTERSTRING, or a compound string constructed (in order) from CAP_PRINTERSTRING, CAP_PRINTERINDEX and CAP_PRINTERSUFFIX.

CAP_PRINTERSTRING specifies the base message to be printed. For compound strings, the CAP_PRINTERSTRING serves as the prefix to the CAP_PRINTERINDEX.

CAP_PRINTERSUFFIX is only available for compound strings, and describes the text (if any) that is to follow the CAP_PRINTERINDEX.

Example of Use:

Consider a Source that supports two CAP_PRINTERs:

TWPR_IMPRINTERTOPBEFORE
TWPR_IMPRINTERBOTTOMBEFORE

The Application then:

- uses CAP_PRINTER to discover the two printers
- sets CAP_PRINTER to TWPR_IMPRINTERTOPBEFORE
  - ✓ sets CAP_PRINTERENABLED to TRUE (turning this printer on)
  - ✓ sets CAP_PRINTERMODE to TWPM_SINGLESTRING
  - ✓ sets CAP_PRINTERSTRING to a string containing today's date
- sets CAP_PRINTER to TWPR_IMPRINTERBOTTOMBEFORE
  - ✓ sets CAP_PRINTERENABLED to TRUE (turning this printer off)

Note that the value of CAP_PRINTER is not important at the time of scanning, it is the other capabilities that control the imprinter, like CAP_PRINTERENABLED; CAP_PRINTER only selects the current printer under negotiation.

B

TWAIN Technical Support

Chapter Contents

E-Mail Support

Developers who are connected to AppleLink and the WWW or Internet have access to TWAIN support groups. The support groups can answer your TWAIN development or marketing questions. There are two support groups: the TWAIN Working Group and the TWAIN Developers distribution.

- The TWAIN Working Group is read by Technical, Marketing and Support representatives from the Working Group companies. You can contact this group via e-mail at twain-wg@twain.org.
- The TWAIN Developers distribution includes TWAIN developers who want to keep up on TWAIN or offer advice to other developers. This distribution includes the TWAIN Working Group. It is the best place to get support because both the Working Group and other developers can respond. You can contact this group via e-mail at twain@twain.org.

TWAIN developers are encouraged to participate on the TWAIN Developer distribution list . All developers responding to questions posted to this distribution should Cc the distribution. The TWAIN Working Group also uses this distribution as a means to communicate with developers. For example, we use the distribution when posting the latest news about TWAIN, asking questions we may have about implementations, and requesting review of any Technical Notes which are under development. Technical Notes provide the mechanism for distributing updated information and corrections to errors that may occur in this document.

Worldwide Web

Developers connected to the WWW can also get on-line information and updates. There is an on-line version of the Developers' matrix with connections to those implementers with WWW pages. In addition, this manual is available as a readable file.

The WWW address is: **http://www.twain.org/**

Information by Fax

From Hewlett-Packard

A short informational white paper on TWAIN and a TWAIN Developer's Toolkit Order Form are available using Hewlett-Packard's fax back system, HP FIRST. To receive these documents call from a touch tone phone or fax machine and the information will be faxed to you.

Phone Numbers:

Inside the US or Canada	800 333-1917
Other locations	08 344-4809

The Document Number is:

3130	TWAIN Toolkit Order Form

Ordering Information

Outside the US and Canada, the TWAIN toolkit is available using the order form available by fax (see above).

Disc Manufacturing, Inc.
A QUIXOTE COMPANY

Introduction to ISO 9660,

what it is, how it is implemented, and how it has been extended.

Clayton Summers

WHO IS DMI?

Disc Manufacturing, Inc. (DMI) manufactures all compact disc formats (i.e., CD-Audio, CD-ROM, CD-ROM XA, CDI, PHOTO CD, 3DO, KARAOKE, etc.) at two plant sites in the U.S.; Huntsville, AL, and Anaheim, CA. To help you, DMI has one of the largest Product Engineering/Technical Support staff and sales force dedicated solely to CD-ROM in the industry.

The company has had a long term commitment to optical disc technology and has performed developmental work and manufactured (laser) optical discs of various types since 1981. In 1983, DMI manufactured the first compact disc in the United States. DMI has developed extensive mastering expertise during this time and is frequently called upon by other companies to provide special mastering services for products in development.

In August 1991, DMI purchased the U.S. CD-ROM business from the Philips and Du Pont Optical Company (PDO). PDO employees in sales, marketing and technical services were retained.

DMI is a wholly-owned subsidiary of Quixote Corporation, a publicly owned corporation whose stock is traded on the NASDAQ exchange as QUIX. Quixote is a diversified technology company composed of Energy Absorption Systems, Inc. (manufactures highway crash cushions), Stenograph Corporation (manufactures shorthand machines and computer systems for court reporting) and Disc Manufacturing, Inc.

We would be pleased to help you with your CD project or answer any questions you may have. Please give us a call at 1-800-433-DISC for pricing or further information.

We have four additional technical papers available entitled

Integrating Mixed-Mode CD-ROM

An Overview to MultiMedia CD-ROM Production

Compact Disc Terminology - 2nd Edition

A Glossary of CD and CD-ROM Terms

These are available upon request
800-433-DISC
302-479-2500
Fax: 302-479-2527

This paper was written in response to the many questions we, as a CD-ROM manufacturer, have received concerning ISO 9660. Our intent was to provide clarity and simplification to a very technical subject. We hope you find it helpful.

For reprinting privileges, call or write to:
Nancy Klocko
Disc Manufacturing, Inc.
1409 Foulk Rd., Suite 102
Wilimington, DE 19803
800-433-DISC, 302-479-2527

Acknowledgments

Without the help of many people, this paper would not have been possible.

Special Thanks go to:

Wendy Upham and Breck Rowell for working on how MS-DOS and the Macintosh implement ISO 9660 and what some of the quirks are

John Sands at Young Minds

Apple Computer

Doug Carson and Associates

and Nancy Klocko and Pam Sansbury for invaluable help editing and revising and Lori Magno for formatting.

Table of Contents

Tables

Figures

Introduction to ISO 9660

The Digital Audio Compact Disc has been called the most successful consumer product ever launched. Since it's introduction in June of 1980, the CD has come to dominate the music industry and become the format of choice for millions of music listeners due to the ultra high fidelity afforded by the digital recording technique and the near indestructibility afforded by the optical design. These same features make the CD very attractive as a carrier of other types of digital information. Another feature that makes the Compact Disc attractive as a medium for digital information is that CDs can be manufactured in large quantities quickly and inexpensively. Also, due to the industry standards defined by the Red Book, Yellow Book, and ISO 9660, any CD can be used on almost any hardware/software platform. It, therefore, comes as no surprise that this 15 grams of poly-carbonate and aluminum which contains billions of bits of data would be embraced by the computer industry to store and distribute huge amounts of data. However, creating a disc that works on multiple platforms is not as simple as copying files to a floppy.

Background

Before ISO 9660, all CD-ROM discs could be read by all CD-ROM drives; however, CD-ROM drives were not supported by any readily available computer operating system. Application developers were required to have software device drivers for each computer and CD-ROM drive combination that they wanted to support. In addition, most applications require a file structure and this had to be provided by each developer as well.

Typically, application developers used a systems house to provide device drivers and file system software as well as build and retrieval engines. The result was that application developers requested both enhancements to the build and retrieval engines and support for additional drive types. The systems houses had to spend critical resources to develop these drivers when they would have preferred to spend those resources in other areas. A committee called High Sierra was formed to develop an industry standard to address the file system software.

The High Sierra proposal was designed to enable data interchange between computers using standardized software. When a computer is equipped for either High Sierra or ISO 9660, data on any properly encoded CD-ROM may be read using standard operating system instructions such as list directory, open, read and close. This reduces the amount of effort required to bring an application to market. In addition, discs may be read by any drive that has standard driver software.

High Sierra was defined and submitted to the **I**nternational **S**tandards **O**rganization in May of 1986. During the time it was being debated and approved, the companies involved in creating the High Sierra proposal went ahead and implemented High Sierra. ISO 9660 was published in April of 1988. ISO made several minor changes during the process that made the ISO 9660 standard incompatible with the High Sierra proposal. The changes involved rearranging the order of the information in the directory record and changing the code that identifies the disc as a High Sierra or ISO 9660 disc, among other, more esoteric, items.

To accurately and repeatedly create ISO-9660 discs requires an understanding of what ISO-9660 is and how it is implemented by different platforms. First a little background information will be presented that helps put the concept of a common format for data interchange into perspective. This will be accomplished by discussing file systems and how they relate to the computer's operating system. Then ISO-9660 will be covered in general terms and some of the more commonly used features and data structures will be covered in some detail. A description, at a conceptual level, of how some operating systems implement support for ISO-9660 and notes regarding some of the peculiarities this causes will then be presented. Then some of the ways ISO-9660 has been extended to provide better support for two particular operating systems, the Macintosh and UNIX environments, will be introduced. Finally, the Frankfurt proposal, an extension to ISO 9660 that allows updating information on a recordable CD, will be discussed.

File Systems

Most operating systems store information in both fast, short term memory usually referred to as Random Access Memory or RAM, as well as in relatively slow, long term memory. Typically, the slow, long term memory takes the form of a floppy disk, or hard disk and is stored as files. If we compare this to someone's office, the fast, short term memory can be compared to the desktop, where things are actually being worked on. The slow, long term disk can be compared to the file cabinet, where unused items are put until needed. The way the operating system keeps track of where files are located is called the file system. Examples of file systems are MS-DOS's FAT (File Allocation Table), the Macintosh HFS (Hierarchical File System), OS/2's HPFS (High Performance File System), and the UNIX File System. All of these file systems are specific to, and optimized for, a particular operating system. ISO-9660 is also a file system. However, it was designed to be independent of any operating system, and because it was designed for CD-ROM, is also "read only". This means that unlike the other file systems mentioned, it does not provide any way to add to or change the information in it. Since ISO 9660 was intended to be used on a diverse group of operating systems, it includes only the minimum information that can be utilized by the widest variety of systems.

Overview of ISO-9660 structure.

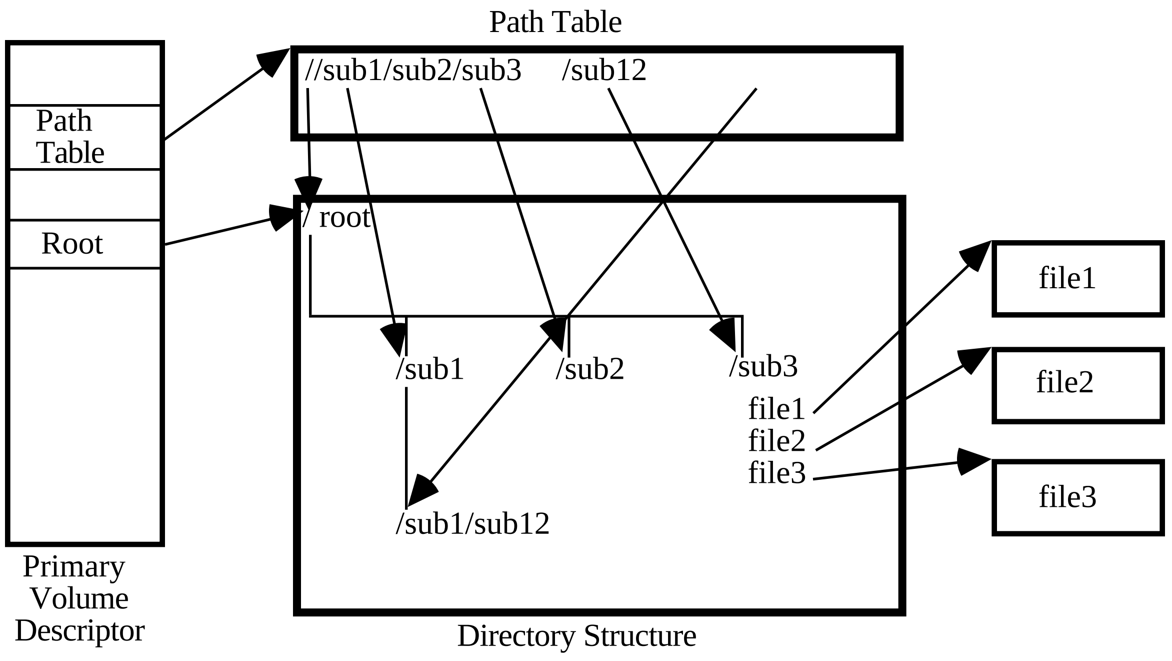


Figure 1. ISO 9660 structures

ISO 9660 data structures fall into three main categories: the Volume Descriptors, the Directory Structures, and the Path Tables. These structures are interrelated as shown in figure 1. The Volume Descriptor tells where the directory structure and the Path Table are located, the directories tell us where the actual files are located, and the Path table gives us short cuts to each directory .

The Volume Descriptors

There are currently four types of Volume Descriptors defined in ISO 9660. Only one of these, the Primary Volume Descriptor, is commonly used. The other types are the Boot Record, the Supplementary Volume Descriptor, and the Volume Partition Descriptor. The Boot Record can be used for systems that must perform some type of initialization before the user can access the volume, although ISO 9660 does not specify what information must be in the Boot Record or how it is to be used. The Supplementary Volume Descriptor can be used to identify an alternate character set for use by systems that do not support the ISO 646 character set. The Volume Partition Descriptor can be used to logically divide the volume into smaller volume partitions, although ISO 9660 does not specify how to do this, only that it can be done.[1] The Volume Descriptors are recorded starting at Logical Sector 16 (which corresponds to two seconds and sixteen sectors into the CD, or in CD "Atime", 00:02:16).

The Primary Volume Descriptor

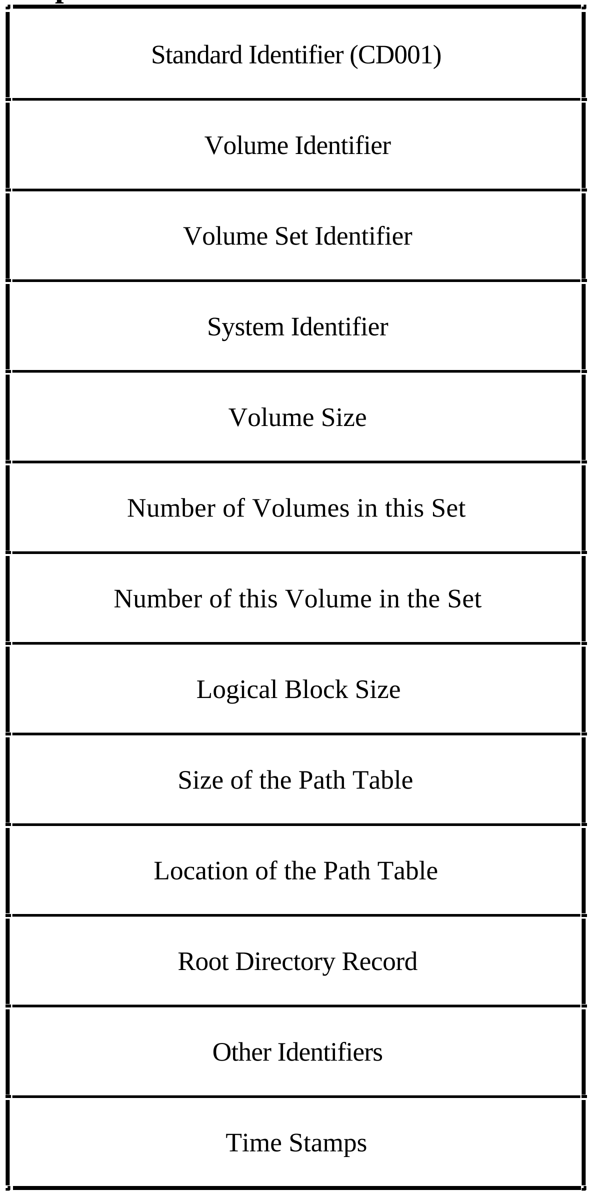


Figure 2. The Primary Volume Descriptor

The Primary Volume Descriptor as seen in figure 2 is the starting point in identifying a CD-ROM. It contains the Standard Identifier, the Volume Identifier, the Volume Set Identifier, the System

[1]ISO 9660-1988, pp. 11, section 8.1.1

Identifier ,the size of the Volume, the number of Volumes in the Volume Set it belongs to, the sequence within the Volume Set that this Volume belongs, the Logical Block size of the blocks in this volume, the size of the Path Table, the location of the Path Table, the Directory record for the Root Directory, other identifiers and important times relating to the Volume.[2]

The Standard Identifier is a set of characters, defined by ISO 9660 to be `CD001`, that tells the Operating System that this is an ISO 9660 disc. This is to distinguish the volume from other file systems that use a similar layout, such as High Sierra, whose Standard Identifier is CDROM, and Compact Disc Interactive, whose Standard Identifier is CD-I.

The Volume Identifier is simply the name that is given to the ISO 9660 volume.
The characters that can be used in the Volume Identifier are restricted to what ISO 9660 calls d-characters and the length is restricted to 31 characters. The d-characters are shown in figure 3.

```
A B C D E F G H I J K L M N O P Q R S
T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 _
```

Figure 3. The d-characters

Some systems, such as the Macintosh, use the Volume Identifier extensively. Others, such as MS-DOS, use it somewhat, and some, such as UNIX, barely use it at all.

The Volume Set Identifier is the name given to the Multiple Volume Set that this Volume belongs to. Like the Volume Identifier, it is restricted to the d-characters and cannot be more than 31 characters long. For example, if this Volume where named DICTIONARY_E_H, it might have a Volume Set Identifier of DICTIONARY, meaning that this Volume contains the words starting with the letter E through the letter H, and the Volume Set is the set of discs for the entire alphabet.

[2]see Appendix A: Table 14 and ISO 9660-1988, pp. 12-16, section 8.4

The System Identifier identifies a system that can recognize and act on logical sectors 0 through 15.[3] While ISO 9660 specifies that this is what the System Identifier is used for, it does not specify what is in sectors 0 through 15, nor does it specify how the data is used. The characters that can be used in the System ID are what ISO 9660 calls a-characters and the length is restricted to 31 characters. The a-characters are shown in figure 4.

```
A B C D E F G H I J K L M N O P Q R S
T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 _ sp
! " % & ' ( ) * + , - . / : ; < = > ?
```

Figure 4. The a-characters

The Volume Size is a number that tells the operating system how many Logical Blocks are in this Volume. A Logical Block is the basic way of locating things in the Volume. All locations are given as Logical Block Numbers. If the Volume is pictured as an Interstate highway, then the Logical Block Numbers are the mile markers.

Volume Set Size is a number that tells the operating system how many volumes are in the Volume Set to which this Volume belongs. The Volume Sequence Number is the place within a multiple volume set that this volume belongs. For example on a disc with Volume Set Size of 5 and Volume Sequence Number of 3, this disc is the third disc of a five disc set.

The Logical Block Size is the number of bytes that make up the smallest amount of space that is allocated in this volume. This number can be 512, 1024, or 2048 bytes. Most ISO 9660 discs use a Logical Block Size of 2048, the same as the Sector Size.

The Path Table Size tells the operating system how many bytes are in the Path Table. Most operating systems that use the Path Table keep it in fast, local memory (RAM), and this number is a quick way

[3]ISO 9660-1988, pp.13, section 8.4.5

for the operating system to know how much memory it needs to allocate before it reads the Path Table. This way the Operating system only reads the Path Table once, saving time. The location of the Path Table must be in the Primary Volume Descriptor since the Path Table itself may be anywhere in the Volume.

The Root Directory record contains the information the operating system needs to locate and read the top level directory. It is formatted exactly the same as any other directory record.[4]

Other identifiers in the Primary Volume Descriptor contain information about who published this Volume, who prepared the data, what the application is, and what the names of the files are that contain the copyright notice, the abstract, and the bibliography.

The time stamps are fields in the Primary Volume Descriptor that contain information about when the Volume was created, when it may have been modified, when the data becomes effective, and when the data becomes obsolete.

[4]See Appendix A, Table 15, page IV

The Directory Structure

The ISO 9660 directory structure is organized in a hierarchical manner similar to most modern file systems.[5] At the top of the hierarchy is the Root Directory, the location of which is identified in the Primary Volume Descriptor. When drawn hierarchically, the directory structure resemble the roots of a tree, with the Root directory at the top of the structure, as shown in figure 5.

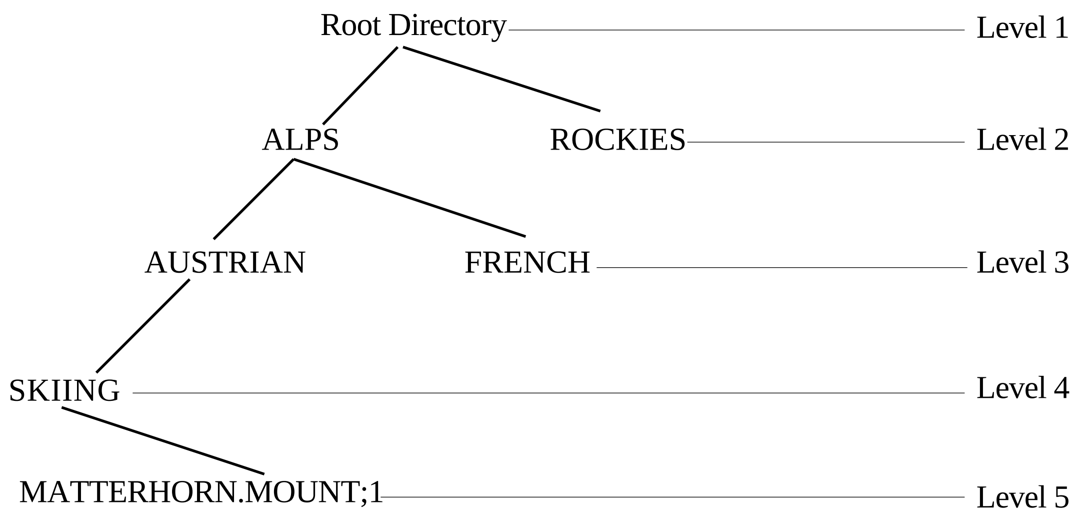


Figure 5. The Directory Hierarchy

[5]ISO 9660:1988, pp.7, section 6.8.2

As shown in figure 5, there are distinct levels in this hierarchy. The Root Directory is the only directory at level 1. In the example illustrated by figure 5, Subdirectories ALPS and ROCKIES are at level 2, Subdirectories AUSTRIAN and FRENCH are at level 3, Subdirectory SKIING is at level 4, and the file MATTERHORN.MOUNT;1 is at level 5. To insure compatibility, ISO 9660 imposes a limit of eight levels to the depth of the directory structure.[6] It also imposes a limit on the length of the path to each file. The length of the path is the sum of the lengths of all relevant directories ,the length of the File Identifier, and the number of relevant directories. The length of the path cannot exceed 255. Again, in figure 5, the sum of the length of the File Identifier ,the lengths of the relevant directories , and the number of relevant directories is 39 as shown in table 1.

Table 1. Length of the Path

Identifier	Length
ALPS	4
AUSTRIAN	8
SKIING	6
MATTERHORN.MOUNT;1	18
number of directories	3
sum of lengths and number of directories	39

A directory in an ISO-9660 volume is recorded as a file containing a set of directory records. Each directory record describes a file or another directory. Every directory has a parent directory. The parent directory contains the directory record that identifies that directory, as shown in figure 6.

[6]ISO 9660:1988, pp.7, section 6.8.2.1

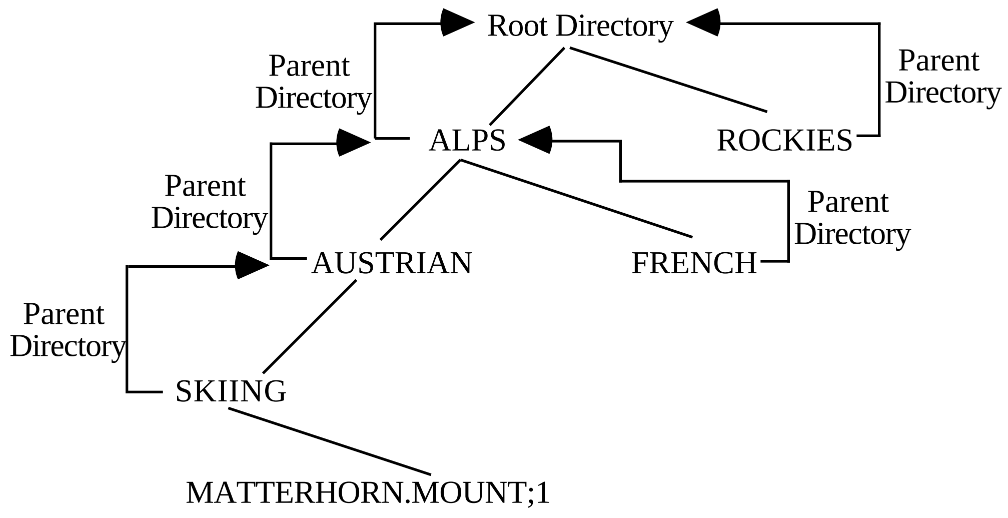


Figure 6. Parent Directories

The Root directory's parent is the Root directory itself.

Each directory also contains a record for its parent directory. A given directory may contain entries for several files as well as for several directories, all of which have the same parent.

File Names

Every file and directory in an ISO 9660 Volume has a name an identifying name associated with it. This name is called the File Identifier. An ISO 9660 File Identifier actually consists of five parts as shown in table 2.[7]

Table 2. The File Identifier

	1) File Name	2) SEPARATOR 1	3) File Name Extension	4) SEPARATOR 2	5) File Version Number
contents	d-characters (see figure 3)	.	d-characters (see figure 3)	;	a number from 1 to 32767
file 1	MATTERHORN	.	MOUNT	;	1
file 2	PIKES_PEAK	.		;	1
file 3		.	HILLS	;	1
directory	SKIING				

The File Identifier must also meet the following conditions:[8]

- If the File Name has no characters, then the File Name Extension must have at least one character, as shown in table 2, file 3.

- If the File Name Extension has no characters, then the File Name must have at least one character, as shown in table 2, file 2.

- The sum of the lengths of the File Name and the File Name Extension cannot exceed 30.

[7]ISO 9660:1988, pp. 10, section 7.5.1
[8]ISO 9660:1988, pp. 10, section 7.5.1

An ISO 9660 directory name is limited to only a Name; it cannot have a SEPARATOR 1 (.), an Extension, a SEPARATOR 2 (;) or a version number, as shown in table 2, directory.[9]

Order of Directory Records

ISO 9660 also specifies the order of the records in a directory.[10] They must be sorted by the relative value of the File Identifier field. Table 3 shows a set of unsorted File Identifiers.

Table 3. File Identifiers

1) File Name	2) SEPARATOR 1	3) File Name Extension	4) SEPARATOR 2	5) File Version Number
MATTERHORN	.	MOUNT	;	1
PIKES_PEAK	.		;	1
	.	HILLS	;	1
SKIING				

The relative value of two File Identifiers is determined in the following way:

- If two File Names have the same content in all byte positions, then these two file names are said to be equal in value.

- If two File Names do not contain the same number of byte positions, the shorter File Name shall be treated as if it were padded on the right with all padding bytes set to (20) (the SPACE character) and as if both File Names contained the same number of byte positions.

[9]ISO 9660:1988, pp. 11, section 7.6.1
[10]ISO 9660:1988, pp.21-22, section 9.3

- After any necessary padding to make the two File Names the same length, the corresponding characters are compared, starting with the first byte position, until a byte position is found that does not have the same character in both File Names. The greater File Name is the one that contains the character whose ASCII value is greater. Table 4 shows the File Names from Table 3 and their relative values.

Table 4. Relative Value of File Names

1) File Name	ASCII Code of first character	Relative value
PIKES_PEAK	80	3
MATTERHORN	77	2
	32	1
SKIING	83	4

The File Name Extensions and Directory Names are also sorted in this manner. The File Version Number is sorted by padding the values with (30) (ASCII "0") on the left to make the lengths equal, then comparing them as above. Table 5 shows the Extensions and Version Numbers from table 3 and their relative values.

Table 5. Relative Value of Extensions and Version Numbers

3) File Name Extension	ASCII Code of first character	Relative Value	5) File Version Number	ASCII Code of first character	Relative Value
MOUNT	77	3	1	31	2
	32	1	1	31	2
HILLS	72	2	1	31	2
	32	1		30	1

The Directory records are then sorted as follows:

- First in ascending order relative to the File Name (or Directory Name)
- Second in ascending order relative to the File Name Extension.
- Third, in descending order relative to the File Version Number.
- Forth, in descending order relative to the value of the Associated File Flag in the File Flags Field. (The associated file comes first, then the file it is associated with).
- Last, in the order of the File Sections of the file. (Only valid if the file is recorded in interleaved mode).

following the above rules, the example started in table 3 is sorted as shown in table 6.

Table 6. Sorted File Identifiers

1) File Name	2) SEPERATOR 1	3) File Name Extension	4) SEPERATOR 2	5) File Version Number
	.	HILLS	;	1
MATTERHORN	.	MOUNT	;	1
PIKES_PEAK	.		;	1
SKIING				

The Path Table

The Path Table indicates to the operating system a short cut to each directory on the disc rather than making the operating system read through each directory to get to the file it needs. This is done primarily to enhance performance. For each directory other than the Root directory, the path table contains a record that identifies the directory, its parent directory, and its location[11].

[11]See Appendix A, Table 16, page VII

Most operating systems read the Path Table once and keep it in memory, rather than reading it over and over again. In the example shown in figure 5, page 9 (the Directory Hierarchy), a system that does not make use of the path table would have to read the root directory to find the location of the ALPS directory, then read the ALPS directory to find the location of the AUSTRIAN directory, then read the SKIING directory to find the location of the file MATTERHORN.MOUNT;1. By making use of the Path Table, the operating system can look up the location of the SKIING directory in the Path Table, read the SKIING directory and find the location of the file. This requires only one seek on the CD-ROM, rather than four. The time difference, for a typical drive with a seek time of 250 msec, is 3/4 of a second. When accessing many files, this difference can significantly affect performance.

Levels of Interchange

ISO 9660 defines three nested levels of interchange which affect the length of the File Identifiers and whether the files must be contiguous or not. Level 1 imposes the most restrictions above and beyond what is specified in ISO 9660. Level 2 impose fewer restrictions, and Level 3 imposes none beyond what is specified in ISO 9660.

Level 1:

- each file shall consist of only one File Section. This means that the files must be contiguous.
- a File Name cannot contain more than eight d-characters or d1-characters.
- a File Name Extension cannot contain more than three d-characters or d1-characters.
- a Directory Identifier cannot contain more than eight d-characters or d1-characters.

An example of Level 1 would be an ISO 9660 disc for the MS-DOS environment, restricted to Level 1 interchange to accommodate MS-DOS's file naming limitations. An important point to note here is that the Level 1 restrictions are more restrictive than the MS-DOS naming conventions. File

Identifiers are still limited to the d-characters as shown in table 1, and Directory Identifiers are limited to d-characters and cannot have an extension.

Level 2:

- each file shall consist of only one File Section. This means that the files must be contiguous.

An example of Level 2 would be an ISO 9660 disc for the Macintosh and UNIX environments, restricted to Level 2 interchange, to allow longer file names.

Level 3:

no restrictions apply beyond what is specified by ISO 9660.

An example of Level 3 would be a CD-ROM-XA disc, which has interleaved data and audio files.

ISO 9660 Implementation Requirements

ISO 9660 also defines requirements for systems that originate or create ISO 9660 volumes and for systems that receive or read ISO 9660 volumes. The requirements for the originating system are primarily of interest to people writing pre-mastering software and will be only briefly mentioned. Pre-mastering is the actual process of creating an ISO 9660 volume. In general, an originating, or pre-mastering system must be able to record a set of files and all of the descriptors described by ISO 9660. The originating system may, however, be restricted to one of the Levels of Interchange. Anyone interested in this level of understanding needs to read the actual ISO 9660:1988 specification.

The receiving system is the system that reads an ISO 9660 disc and makes the data accessible to the user. The receiving system is expected to be able to read the files and descriptors on a volume that meets at least one of the above interchange levels. It is also expected to make the information in these files available to the user. All of them except for the Associated Files, which we will discuss in more detail later. The receiving system is also expected to make available to the user much of the information in the Volume Descriptors and Path Table. All of the following information should be visible to the user:

From the Primary Volume Descriptor:

- Volume Identifier
- Volume Set Identifier
- Copyright File Identifier
- Abstract File Identifier
- Bibliographic File Identifier

From each Supplementary Volume Descriptor:

- Volume Identifier
- Bit 0 of the Volume Flags field, which indicates if the escape sequences used are registered
- Escape Sequences, which define the d1-character set
- Volume Set Identifier
- Copyright File Identifier
- Abstract File Identifier
- Bibliographic File Identifier

From each Path Table Record:

- Parent Directory Number
- Directory Identifier

From each Directory Record:

- File Name of a File Identifier
- File Name Extension of a File Identifier
- Directory Bit of the File Flags field

Similar to the Levels of Interchange, which apply to the originating system, ISO 9660 also defines Levels of Implementation, which apply to the receiving system:

Level 1 implementation:

- At level 1, the receiving system does not have to make any of the data contained in and pointed to by the Supplementary Volume Descriptors available to the user. Most implementations are Level 1.

Level 2 implementation has no such restrictions.

Implementations of ISO 9660

As figure 7. shows, each operating system has its own specific file system. Independent of the operating systems is ISO 9660.

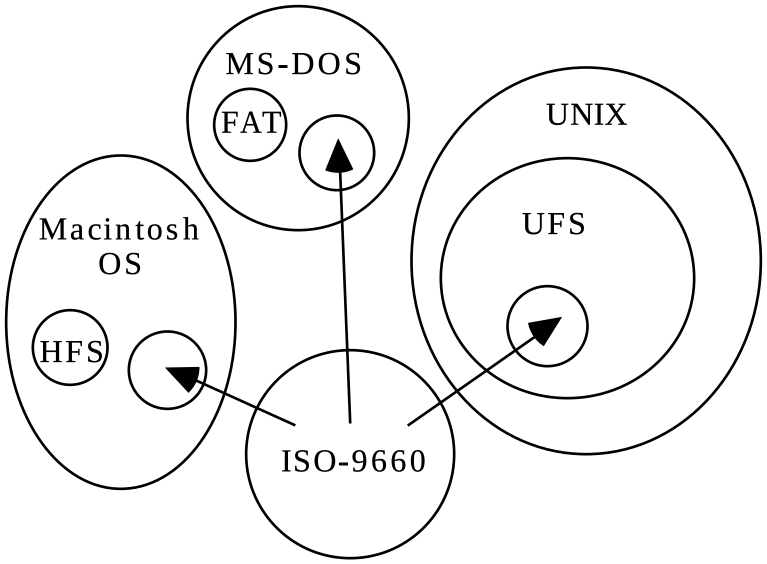


Figure 7. ISO 9660 World View

In order for a given platform to transparently implement ISO-9660, the operating system must convert the ISO-9660 file system into something that looks like its normal file system. This way, the application programs can use the ISO-9660 disc as if were a native read only file system.

DOS

Under MS-DOS, a program called Microsoft CD-ROM Extensions (MSCDEX.EXE) intercepts operating system calls to the CD-ROM and makes the ISO-9660 volume look like a normal, read only, hard disk. However, MS-DOS does not support File Version Numbers, which are part of an ISO 9660 File Identifier. To properly handle the File Version number, when MS-DOS requests a File Identifier from the ISO-9660 volume, Extensions modifies the File Identifier it returns to the operating system by stripping off the File Version Number and only returning the file with the highest version number. To the operating system and the user, the ISO-9660 disc is accessed using a drive letter, just like a write protected hard disk or floppy disk, or network drive.

When creating discs for the DOS environment, the biggest pitfall that developers encounter is that the ISO 9660 file naming conventions. ISO-9660 is much more restrictive than DOS as to the characters that may be used, but much less so as to the length of the names.

MS-DOS file names are limited to eight characters in the file name, and 3 characters in the file name extension. ISO-9660 File Identifiers can be up to 30 characters, and does not specifically limit the length of the Name or the Extension. If the pre-mastering system is DOS based, the length is not a problem, but if the disc is being pre-mastered on a Macintosh or UNIX machine, for example, the file names may be longer than DOS allows. When this disc is read on a DOS machine, the names appear to be truncated, but DOS cannot access the files, as shown in table 7.

Table 7. Long ISO File Identifiers under MS-DOS

Original ISO-9660 File Identifier	File Name as seen by MS-DOS	Response when accessed
MATTERHORN.MOUNT;1	MATTERHO.RNM	File Not Found
EVEREST.MNT;1	EVEREST.MNT	OK

MS-DOS allows a much larger group of characters to be included in file names than does ISO -9660. If an ISO-9660 Volume is made using characters that are allowed under MS-DOS, but do not conform to the d-character set, as shown in figure 2, there is a good chance that some files will not be readable. The files appear normally in the directory, but DOS may or may not be able to open all of the files. This problem is particularly perplexing since the file with the illegal character can be read, but the next file after it may not be. This problem occurs if you have a set of files, which have the same names up to some point, after which one file has an illegal character, and one file a SEPARATOR 1 (period, "."), as demonstrated in table 8.

Table 8. Illegal d-characters and Microsoft extensions

`INSTALL-1.TXT;1`	Reads OK
`INSTALL.BAT;1`	Reports "File not found"
`INSTALL.EXE;1`	Reports "File not found"
`INSTALL.TXT;1`	Reports "File not found"

In the example shown in table 10, the first name has an illegal ISO-9660 character, the dash (`-`). This file is readable. However, when attempting to read any of the other three files, MS-DOS reports "File not found". This appears to be occurring because Microsoft Extensions assumes that the directory records are sorted properly and when it sees a name that, if sorted properly, would come after the name it is searching for, it stops searching and proclaims that it cannot find the file. In the example shown in table 9, if MS-DOS is searching for `INSTALL.BAT`, it reads the first File Name and compares it to `INSTALL.BAT`. If properly sorted according to the rules specified in ISO 9660, INSTALL.BAT would appear before INSTALL-1.TXT, as shown in table 11. However, the actual order of the records has INSTALL-1.TXT first, before any of other INSTALL files. Therefore, when MS-DOS searches for INSTALL.BAT, it sees INSTALL-1.TXT, which should come after INSTALL.BAT, and comes to the conclusion that INSTALL.BAT is not there.

Table 9. Sorting illegal ISO-9660 File Identifiers

Order of directory records in ISO-9660 Volume	Correctly sorted directory records
`INSTALL-1.TXT;1`	`INSTALL.BAT;1`
`INSTALL.BAT;1`	`INSTALL.EXE;1`
`INSTALL.EXE;1`	`INSTALL.TXT;1`
`INSTALL.TXT;1`	`INSTALL-1.TXT;1`

What appears to be causing this is that most pre-mastering packages do not sort the names properly **IF** they are told to put illegal d-characters in the ISO 9660 volume. In particular, they appear to no longer sort the File Name and the File Name Extension separately, but treat the entire File Identifier as the File Name. The best way to avoid this is to not have any illegal d-characters in the File Names, Directory Names, or File Name Extensions.

Macintosh

The Macintosh supports ISO 9660 by adding an extension onto the operating system, called the foreign file access extension. This operating system extension, along with code that tells it how to convert ISO-9660, makes an ISO-9660 disc appear on the desktop just like any other write-protected HFS Volume, with its own icon on the Macintosh desktop.

Even though the ISO 9660 disc looks like a normal volume, not all of the data that the Macintosh needs to display the volume on the desktop is available in the ISO 9660 data. The additional information gets created by the ISO-9660 access software when an ISO directory is opened. The result of this is that the user has no control over the placement of folders and files, and an ISO 9660 disc loses some of the look and feel so important to the Macintosh.

Another problem that occurs with ISO 9660 on the Macintosh has to do with how the Macintosh file system implements executable files, or applications as they are known to Mac users. All of the files on a standard ISO 9660 disc will appear on the desktop as generic documents. To correct this deficiency, Apple uses the reserved system use field in the directory record and associated files in ISO 9660 to add extra information that allows applications to run from an ISO 9660 disc. See the discussion on extensions to ISO 9660 for more details, page 29.

For some developers, another source of difficulties is the File Version Number. The File Version Number is added to each file when it is pre-mastered to make the File Identifiers comply with the ISO specification. Applications that worked from the hard disk may not work from the CD-ROM for the simple reason that they are trying to find a file whose name now has a ';1' on the end of it.

UNIX

UNIX systems typically support ISO 9660 by including the programs into the operating system, rather than using external conversion programs or extension drivers. This is normally done by recompiling the operating system along with the new code to support CD-ROM and ISO 9660. The CD-ROM can then be "mounted" on an existing directory in the UNIX directory structure. On UNIX style systems, instead of having different drive letters or volumes, different drives, or devices, are "mounted" on a directory. This means that each UNIX system only has one directory structure, and all devices are part of that structure as shown in figure 8.

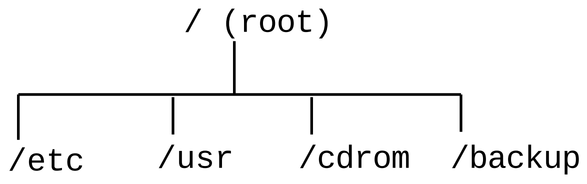


Figure 8. UNIX directory

In this example, the `etc` directory may be on the hard disk with the operating system, the `usr` directory may be a second hard disk used to store user's files, the `cdrom` directory may be an ISO-9660 disc, and the `backup` directory may be a network drive on an entirely different system.

Unfortunately, because of the way UNIX has historically been implemented, there is quite a bit of variation among UNIX systems as to how they support CD-ROM and how the files on the CD-ROM will appear to the user. As there are well over 100 UNIX compatible Operating Systems currently available, no attempt will be made to address each one separately. The best way to find out how a system will react is to read the manual pages for the mount command and for cdrom and the CD-ROM file system. Some of the most common idiosyncrasies will be mentioned, however.

Some UNIX systems do not modify the file names at all and the user will see upper case files names, with the version number appended. This seems reasonable, since it shows the complete File Identifier, but on many systems the semi-colon character has special meaning and causes access difficulties. Other systems will strip off the version number in much the same way that MSDOS does, and have the option to either convert the characters to lower case or to leave them uppercase. Of the systems that have this option, some default to converting to lower case, while others default to leaving everything upper case. See table 10 for examples of what these different options look like.

Table 10. UNIX File name conversions

ISO 9660 File Identifier	Conversion being applied	File Name as it appears to user
`MANPAGES.1;1`	none	`MANPAGES.1;1`
`MANPAGES.1;1`	no version number	`MANPAGES.1`
`MANPAGES.1;1`	lower case, no version number	`manpages.1`

This provides the system administrator with a lot of flexibility, but makes it difficult for applications that have file names "hard coded" into the programs. These discs have to include clear instructions to the user to have the system mount the disc with filenames in the proper case. Anyone who has been involved in porting applications to different UNIX systems will find this situation not at all unusual.

Extensions to ISO 9660

ISO 9660 works well for a majority of operating systems. In some cases, however, it has proven to be difficult or impossible to use. To make it more usable in their respective environments, Apple and the UNIX community established extensions to ISO 9660. The Apple Extensions were created by Apple and are generally known as Apple ISO 9660. The UNIX community assembled a group of people created what is known as the Rock Ridge Proposal.

Recordable CD, or CD-R, opens up the possibility of updating information already on a CD-ROM. ISO-9660 was never intended to support this function. To meet this new requirement, a group met at Frankfurt, Germany and developed an update to ISO 9660. This update is called the Frankfurt Group Proposal - Volume and File Structure for Read-Only and Write-Once Compact Disc. At this point in time, the Frankfurt Proposal has been approved by the European Computer Manufacturers Association as standard ECMA 168. The Frankfurt Proposal is very complex and difficult to implement. To simply provide a way to update information on an ISO 9660 volume, a much simpler method has been implemented by several companies. This method is commonly known as Updatable ISO 9660 or Multi-Session ISO 9660.

Apple ISO 9660[12]

The Macintosh operating system requires a lot of specialized data to support its graphical user interface. The information needed to implement these features is stored in two places by the HFS file system. Some information is stored in special fields in the HFS directory record. Most of the data is stored in the file itself, in a specially formatted area called the resource fork. The data in the resource

[12] For further information, see "CD-ROM & the Macintosh Computer, A Gentle, Technical Introduction to Creating CD-ROMS for the Apple Macintosh computer family" and "The Apple CD-ROM Handbook"

fork is normally only accessible through system calls to a part of the Macintosh Operating system known as the Resource Manager. The resource fork contains things like the Menu layout, the Window definitions, user preferences, and text messages that the application displays, and the actual executable code in the case of an application. The remaining data in a file is called the data fork. The data fork is the same as a DOS or UNIX file.

Most Macintosh files have both forks, a data fork and a resource fork. ISO 9660 accommodates the resource fork very well through the use of the associated file. When a Macintosh file is recorded in an ISO volume, the resource fork is recorded as an associated file, and the data fork is recorded as a normal file.

Unlike the resource fork, there is some information vital to the Macintosh environment that can not be stored on a standard ISO 9660 volume. The native Macintosh file system, HFS, stores information regarding the file icon's position on the screen, what the icon looks like, what type of file it is, what application created it, and file attributes. The file attributes contain information such as if the file is visible, if the file is locked, and if it is an alias. To capture this information on an ISO 9660 volume, Apple created an extension to ISO 9660, making use of the System Use Field in the Directory Record. This field is allocated in ISO 9660, but how it is used is left open.

Apple was not, however, able to make an ISO disc look exactly like an HFS disk. Currently, there is no way to record the positions of icons on the desktop in an ISO 9660 volume, so these are created when a directory is opened, with no way to control where they are located. Also, because of the way the Finder works, files and folders on an ISO disc only appear with generic icons, as seen in figure 9.

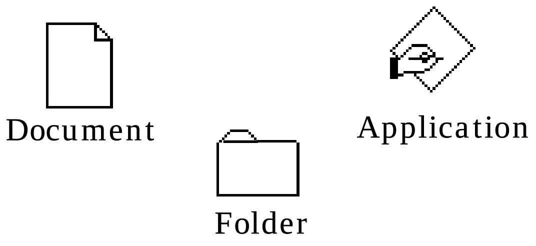


Figure 9. Apple Macintosh generic Icons

These generic icons are displayed even if the correct icons are in the Apple extensions area because the Finder assumes there is a desktop database from which to retrieve these icons and uses a special call to retrieve them. This part of the Finder was written to be very HFS specific and does not work with ISO 9660 volumes even if there is a file called desktop in the volume. If these files are copied to an HFS volume, however, the correct icon will be shown. If the Apple extensions are not present, all files will show as generic documents.

The Protocol Identifier

To identify a volume as having the Apple extensions and inform the operating system what type of file name translation to perform, the System Identifier field in the Primary Volume Descriptor is defined to be the following[13]:

`"APPLE COMPUTER, INC., TYPE: "` followed by a four byte identifier.

The four byte identifier tells the system what version of Apple Extensions this is and whether or not to perform automatic file name translation for ProDOS (used on the Apple //GS). A typical identifier, that tells the system not to perform ProDOS translation and that the version number is 2 is `"0002"`.

[13]"CD-ROM & the Macintosh Computer...", pp. 23

The Directory Record System Use Field

The System Use Field[14] of an ISO 9660 Directory record is used to store the additional information needed by the Macintosh Operating system. The additional fields are defined as shown in table 11[15].

Table 11. Apple ISO 9660 Directory Record System Use Field

BP	**Field Name**	**Content**
1 to 2	Signature ID	(41)(41) "AA" Apple signature
3	SystenUse Extension Length	(0E) bytes
4	System Use ID	(02) for HFS
5 to 8	HFS fileType	(MSB-LSB)
9 to 12	HFS fileCreator	(MSB-LSB)
13 to 16	HFS finder flags	(MSB-LSB)

Apple also allows more than one System Use Extension field in a single directory entry, limited only by the size of the directory entry (it cannot be larger than a logical block). Apple intended for the Signature ID to be used to identify extensions for different systems. Each system could then ignore the fields whose Signature ID it did not recognize. This method of sharing the System Use field was later adopted by both the Rock Ridge Group and the Frankfurt Group for extending the ISO 9660 Directory record.

[14] See ISO 9660:1988, pp. 21, section 9.1.13
[15] "CD-ROM & the Macintosh Computer...", pp. 25

The Rock Ridge Proposals

Rock Ridge is a group of companies that began meeting in July 1990 to resolve issues with ISO 9660 that make it difficult to use as a distribution medium for some operating systems (UNIX based systems being their primary concern). Some of the issues addressed include long filenames with lower case characters, directory structures much deeper than the eight levels allowed by ISO 9660, different file types and access privileges. The Rock Ridge proposals offer industry standard solutions for the distribution of data and software on CD-ROM media by extending the ISO 9660:1988 specification while remaining completely compliant with it.

These proposals deal with two main areas. First, it establishes a standardized way for multiple file system extensions to coexist in one ISO 9660 Directory record. I then defines a way to record POSIX[16] files and directories in an ISO volume without modifying their original directory information. POSIX is a standard for a Portable Operating System Interface much of which is based on how the UNIX operating system works. This allows standard UNIX style file names and directories to be recorded in an ISO 9660 volume without any modification.

Rock Ridge System Use Sharing Protocol (SUSP)

The System Use Sharing Protocol provides a standard way for multiple systems to record system specific extensions in the System Use field by defining a generic field format for System Use Fields, and a set of generic System Use Fields for that can be used to:

- continue the System Use Fields in an area outside of the directory record
- do additional padding
- identify that SUSP is being used

[16]Institute of Electrical and Electronic Engineers Portable Operating System Interface IEEE Std. 1003.1-1990

- terminate the SUSP area
- identify which system specific extension is being used.

The System Use Field Format is as shown in table 12.

Table 12. SUSP System Use Field

BP	Field Name	Content
1 to 2	Signature Word	Identifies what type of System Use Field this is
3	Length of System Use Field (LEN_SUF)	bytes
4	System Use Field Version	Version Number of System Use Field
5 to LEN_SUF	Data	Content of System Use Field

The Signature Word identifies what the following system use field will be used for. The Signature Words that are defined by SUSP are:

"CE"	Continuation Area. This field points to the Logical Block Number where the System Use Area is continued. This allows the System Use data to extend beyond a single Logical Block.
"PD"	Padding Field. More than one of this field may appear in any given System Use Area. This is used to fill up empty areas such as might occur at the end of a Logical Block before going on to the Continuation Area.

"SP"	System Use Sharing Protocol Indicator. Only one of this field can be in a Volume. It must be in the System Use Area of the first directory record of Root directory. This identifies the Volume as adhering to the System Use Sharing Protocol.
"ST"	System Use Sharing Protocol Terminator. This field is an optional field to mark the end of SUSP's use of the System Use area.
"ER"	Extensions reference. This field may be mandatory or not, depending on the specification which it describes. This is determined by specifications that make use of the System Use Sharing Protocol.

For more information regarding these System Use Fields, see the System Use Sharing Protocol proposed specification, available on the DMI "demo" disc. The System Use Sharing Protocol does not provide new features by itself, but provides a common base on which to establish new ISO 9660 extensions, such as the Rock Ridge Interchange Protocol.

Rock Ridge Interchange Protocol (RRIP)

Rock Ridge Interchange Protocol was designed to allow users of POSIX and other UNIX like systems to retain much of the directory information that is in the native file system. These systems use directory entries for much more than just pointing to files. Directory entries can point to other entries (symbolic links or aliases) or to device drivers that are linked to peripheral devices such as hard disks, tape drives and CD-ROM drives (device files). The directory entry includes information that lets the system know the file type. Is it a regular file, a directory, a symbolic link, or a device file? The Directory entry also has information regarding who has permission to read, write and execute each file. Most of these systems are multi-user systems, and must be able to limit who can write to the device file that contains the operating system, or it could accidentally (or purposely?) be erased.

File Names

The Rock Ridge Interchange Protocol is intended to be portable across a variety of POSIX compliant systems, so it makes suggestions that increase portability. However, unlike ISO 9660, it does not set hard and fast limits on the character set that can be used in file names and directory names. Systems that support RRIP must, however, treat file names that are the same except for the case of the letters as being different files (a file named `alpine` is not the same as a file named `Alpine`). RRIP suggests that the only the characters in table 13 be used for maximum portability.

Table 13. Suggested Characters for RRIP File Identifiers

Suggested Characters	ASCII Hexadecimal Code
'A' through 'Z"	(41) - (5A)
'a' through 'z'	(61) - (7A)
period .	(2E)
underscore _	(5F)
hyphen -	(2D)

Deep directories

On most POSIX type systems, not only do the file names tend to be long, but the directories tend to get much deeper than the 8 levels allowed by ISO 9660. For this reason, RRIP defines a way to remap deep directories that allows it to be ISO compliant, while at the same time retaining the deep directory structure on systems supporting RRIP.

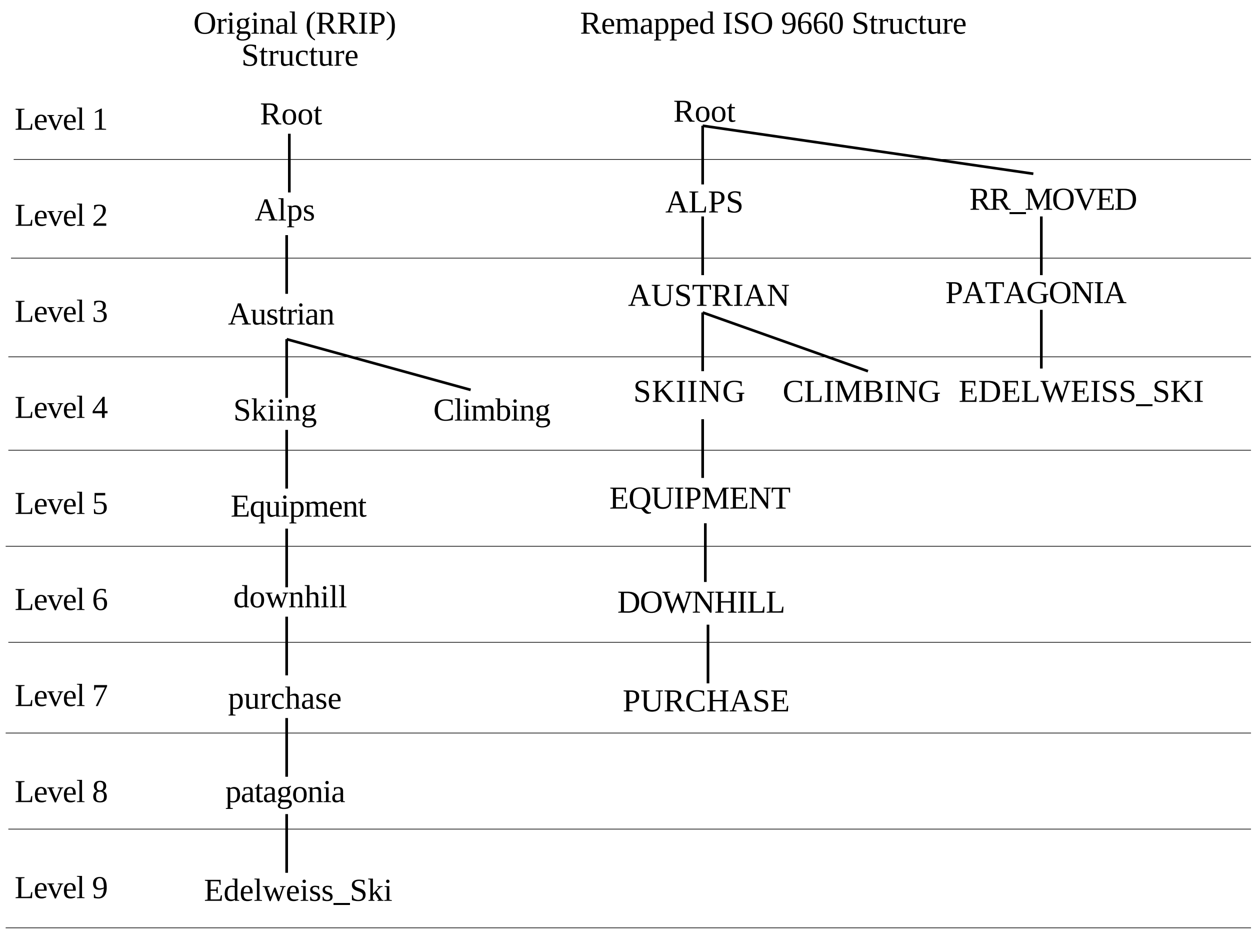


Figure 10. Remapped Directory structure

A directory that was originally at the eighth level is relocated higher up in the directory structure as shown in figure 10. Generally, the pre-mastering software will handle relocating directories automatically. Probably the most common pre-mastering package that supports RRIP, Makedisc from Young Minds, Inc., creates a new directory at the root level, called RR_MOVED, and places all relocated directories here. This directory is only visible on systems that do not support RRIP. On RRIP systems, you see the original, deep, directory structure.

For more information on the exact layout of the RRIP System Use Fields, see the Rock Ridge Interchange Protocol proposed specification, available on the DMI "demo" disc.

Updatable ISO 9660

A simple method for allowing CD-WO discs to be updated has been implemented by several companies. This technique involves writing a new, complete directory structure each time the disc is updated. The new structure is recorded as an ISO 9660 structure, starting at 00:02:16 into the latest recording session, another session being recorded each time the disc is updated. For an explanation of recording sessions and CD-WO in general, see the Compact Disc Terminology paper, available from DMI, and the Philips and Sony Orange book, *Recordable Compact Disc Systems.* A simple volume that has not been updated, and then the same disc after updating is shown in figure 11. The ISO 9660 directory structure in Session 1 only reflects files 1 and 2. The directory structure recorded in session 2, however, reflects not only files 3 and 4, recorded during session 2, but also files 1 and 2, recorded in the session 1.

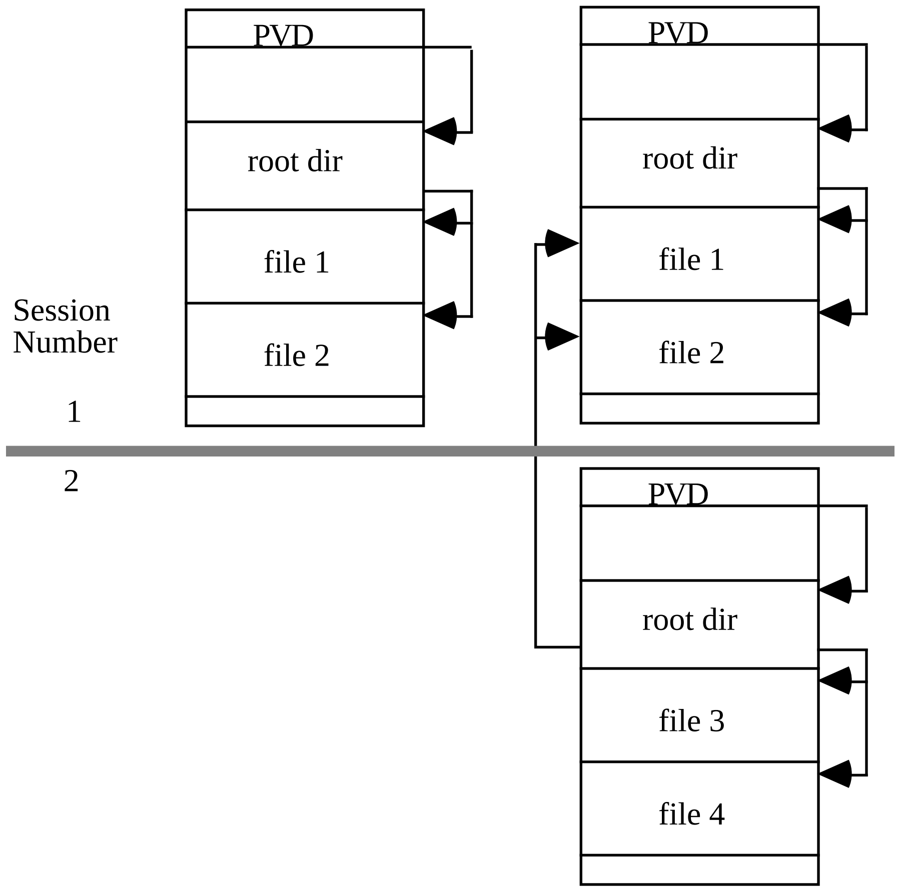


Figure 11. Updatable ISO 9660

In order for a CD-ROM drive to recognize the updated areas, the drive must be able to recognize that there are multiple sessions on the disc, and the ISO 9660 implementation must be able to use the Primary Volume Descriptor from the last session on the disc. This is commonly known as "Multi-Session" compatibility.

The Frankfurt Group Proposal, ECMA 168

The Frankfurt Group is an ad hoc group of companies which share common interests and goals concerning Read Only and Write Once Compact Disc Technology. The Write Once Technology is governed by the Philips and Sony Orange Book, *Recordable Compact Disc Systems.* The Frankfurt Group Proposal covers two types of Volume and File Structures. Type 1 is the original ISO 9660 specification, and is used only for Read-Only discs. Type 2, the Frankfurt Proposal, is an extension to Type 1 that allows incremental recording of a Compact Disc Write Once (CD-WO or CD-R) and updates to be recorded to an existing CD-WO. This proposal is much more complex and flexible than the Updatable ISO 9660. In addition to providing a way to incrementally record and update discs, Type 2 also defines a framework that furnishes the same type of functionality that the Rock Ridge Proposals provide. The specification, however, is not the same as the Rock Ridge Proposals. For more information regarding the Frankfurt Proposal, see ECMA 168 - Volume and File Structure for Read-Only and Write-Once Compact Disc.

Summary of ISO 9660

- An ISO 9660 volume consists of the following data structures:
  - Volume descriptors (what is this volume, and where are the other data structures)
  - The Directory Structure (where are and what are the names of the files and directories)
  - The Path Table (what are the locations and parent directories of each directory)
- ISO 9660 defines three levels of interchange for the Originating System:
  - Level 1 (File names limited to 8.3, directory names limited to 8, and all files must be contiguous) To insure compatibility across the most platforms, restrict the ISO volume to Level 1 Interchange.
  - Level 2 (All files must be contiguous)
  - Level 3 (no limitations beyond the ISO specification itself)
- ISO 9660 defines two levels of Implementation for the Receiving System:
  - Level 1 (Receiving System can ignore any Supplementary Volume Descriptors)
  - Level 2 (Receiving System must utilize all data available)
- Example of ISO 9660 Implementations:
  - MS-DOS:
    - MS-DOS supports ONLY Level 1 Interchange.
    - Access to ISO 9660 volumes is performed by Microsoft Extensions, MSCDEX.EXE.
    - Extensions strips off the version number and only recognizes the highest version.
    - Extensions can not find files and directories with File Identifiers longer than 8.3.
    - Extensions does not handle illegal ISO characters (non d-characters) very well.
  - Apple Macintosh:
    - Apple Macintosh supports Level 2 Interchange. Volumes that must be usable under MS-DOS, as well as Macintosh, must be restricted to Level 1 Interchange.

- Access to ISO 9660 volumes performed by system extensions.
- Layout of folders and files on desktop is created when an ISO 9660 directory is opened.
- ISO 9660 disc must include Apple extensions to run Macintosh applications from the ISO volume.
- ISO 9660 file identifiers include the version number.

- UNIX
  - Most UNIX type systems support Level 2 Interchange. Volumes that must be usable under MS-DOS, as well as UNIX, must be restricted to Level 1 Interchange.
  - Access to ISO 9660 volumes is usually incorporated into the operating system.
  - There is considerable variation between UNIX type implementations as to how the File Identifiers appear to the user.
  - Some systems have options to convert File Identifiers to lower case, and remove the File Version Number, so the same volume can appear different, even on the same machine.

- Extensions to ISO 9660
  - Apple ISO 9660 provides the Macintosh system with additional data needed to launch applications from an ISO 9660 volume.
  - The Rock Ridge Proposals provide a more UNIX like environment for distributing data to a variety of UNIX like platforms.
  - Updatable ISO 9660 provides a simple way to add more data to a previously recorded CD-WO.
  - The Frankfurt Group Proposal, ECMA 168, provides a way to append data to a CD-WO, and provide a more UNIX like environment for distributing data to a variety of UNIX like platforms.

Appendix A: ISO 9660 Structures

Table 14. Primary Volume Descriptor

BytePosition	Field Name	Content
1	Volume Descriptor Type	1
2 to 6	Standard Identifier	CD001
7	Volume Descriptor Version	1
8	Unused Field	(00)[17] byte
9 to 40	System Identifier	a-characters allowed[18]
41 to 72	Volume Identifier	d-characters allowed[19]
73 to 80	Unused Field	(00) bytes
81 to 88	Volume Space Size	Number of logical blocks in the Volume
89 to 120	Unused Field	(00) bytes
121 to 124	Volume Set Size	The assigned Volume Set size of the Volume
125 to 128	Volume Sequence Number	The ordinal number of the volume in the Volume Set
129 to 132	Logical Block Size	The size in bytes of a Logical Block
133 to 140	Path Table Size	Length in bytes of the path table
141 to 144	Location of Type L Path Table	Logical Block Number of first Block allocated to the Type L Path Table, Type L meaning multiple byte numerical values are recorded with least significant byte first. This value is also recorded with least significant byte first.
145 to 148	Location of Optional Type L Path Table	0 if Optional Path Table was not recorded, otherwise, Logical Block Number of first Block allocated to the Optional Type L Path Table.
149 to 152	Location of Type M Path Table	Logical Block Number of first Block allocated to the Type M Path Table, Type M meaning multiple byte numerical values are recorded with most significant byte first. This value is also recorded with most significant byte first.
153 to 156	Location of Optional Type M Path Table	0 if Optional Path Table was not recorded, otherwise, Logical Block Number of first Block allocated to the Type M Path Table.

[17] Numbers surrounded by parentheses () are hexadecimal numbers.
[18] a-characters are A-Z, 0-9, _, space, !, ", %, &, ', (,), *, +, ,, -, ., /, :, ;, <, =, >, ?
see ISO-9660:1988, Annex A, Table 15
[19] d-characters are A-Z, 0-9, _
see ISO-9660:1988, Annex A, Table 14

157 to 190	Directory record for Root Directory	This is the actual directory record for the top of the directory structure. See the section on directory records for the format of this data.
191 to 318	Volume Set Identifier	Name of the multiple volume set of which this volume is a member. d-characters allowed.
319 to 446	Publisher Identifier	Identifies who provided the actual data contained in the files. a-characters allowed.
447 to 574	Data Preparer Identifier	Identifies who performed the actual creation of the current volume. a-characters allowed.
575 to 702	Application Identifier	Identifies the specification of how the data in the files are recorded. For example, this field might contain SGML if the files were recorded according to the Standard Generalized Markup Language
703 to 739	Copyright File Identifier	Identifies the file in the root directory that contains the copyright notice for this volume. If there is no copyright file, this field should contain all spaces (20) Level 1 interchange restrictions apply.[20]
740 to 776	Abstract File Identifier	Identifies the file in the root directory that contains the abstract statement for this volume. If there is no copyright file, this field should contain all spaces (20) Level 1 interchange restrictions apply.
777 to 813	Bibliographic File Identifier	Identifies the file in the root directory that contains bibliographic records. ISO-9660 does not specify the format of these records. If there is no copyright file, this field should contain all spaces (20) Level 1 interchange restrictions apply.
814 to 830	Volume Creation Date and Time	Date and time at which the volume was created. <u>Represented by seven bytes:</u> 1: Number of years since 1900 2: Month of the year from 1 to 12 3: Day of the Month from 1 to 31 4: Hour of the day from 0 to 23 5: Minute of the hour from 0 to 59 6: second of the minute from 0 to 59 7: Offset from Greenwich Mean Time in number of 15 minute intervals from -48(West) to +52(East)
831 to 847	Volume Modification Date and Time	Date and time at which the volume was last modified. Represented the same as the Volume Creation Date and Time
848 to 864	Volume Expiration Date and Time	Date and Time at which the information in the volume may be considered obsolete. Represented the same as the Volume Creation Date and Time
865 to 881	Volume Effective Date and Time	Date and Time at which the information in the volume may be used. Represented the same as the Volume Creation Date and Time

[20] For a description of the level 1 interchange restrictions, see page <?>

882	File Structure Version	1
883	Reserved for future standardization	(00)
884 to 1395	Application Use	This field is reserved for application use. Its content is not specified by ISO-9660.
1396 to 2048	Reserved for future standardization	All bytes must be set to (00).

Table 15. Directory Record

BP	**Field Name**	**Content**
1	Length of directory Record (LEN_DR)	Bytes
2	Extended Attribute Record Length	Bytes - this field refers to the Extended Attribute Record, which provides additional information about a file to systems that know how to use it. Since few systems use it, we will not discuss it here. Refer to ISO 9660:1988 for more information.
3 to 10	Location of Extent	This is the Logical Block Number of the first Logical Block allocated to the file.
11 to 18	Data Length	Length of the file section in bytes
19 to 25	Recording Date and Time	This is recorded in the same format as the Volume Creation Date and Time
26	File Flags	One Byte, each bit of which is a Flag: Bit 0 File is Hidden if this bit is 1 1 Entry is a Directory if this bit is 1 2 Entry is an Associated file is this bit is 1 3 Information is structured according to the extended attribute record if this bit is 1 4 Owner, group and permissions are specified in the extended attribute record if this bit is 1 5 Reserved (0) 6 Reserved (0) 7 File has more than one directory record if this bit is 1
27	File Unit Size	This field is only valid if the file is recorded in interleave mode. Otherwise this field is (00)
28	Interleave Gap Size	This field is only valid if the file is recorded in interleave mode. Otherwise this field is (00)
29 to 32	Volume Sequence Number	The ordinal number of the volume in the Volume Set on which the file described by the directory record is recorded.
33	Length of File Identifier (LEN_FI)	Byte

34 to (33 + LEN_FI)	File Identifier	Interpretation depends on the setting of the directory bit in the File Flags If set to ZERO, then The field refers to a File Identifier, as described below If set to ONE, then The field refers to a Directory Identifier, as described below.
34 + LEN_FI	Padding Field	Present only if the length of the File Identifier is an even number. If present, value is (00)
LEN_DR - LEN_SU + 1	System Use (LEN_SU)	Reserved for system use. If necessary, so that the length of the directory record is an even number of bytes, a (00) byte may be added to terminate this field.

The Path Table Record contains the following fields:[21]

Table 16. Path Table Record

BP	**Field Name**	**Content**
1	Length of Directory Identifier (LEN_DI)	Length in Bytes
2	Extended Attribute Record Length	If an Extended Attribute Record is recorded, this is the length in Bytes. Otherwise, this is (00)
3 to 6	Location of Extent	Logical Block Number of the first Logical Block allocated to the Directory
7 to 8	Parent Directory Number	The record number in the Path Table for the parent directory of this directory
9 to (8 + LEN_DI)	Directory Identifier	This field is the same as in the Directory Record
(9 + LEN_DI)	Padding Field	Present only if LEN_DI is an odd number. (00)

[21] ISO 9660:1988, pp. 22, section 9.4

Appendix B: Common Q&A

1. What do I need to do to make my MS-DOS data ready for ISO-9660?

There are three things that need to be checked to insure that your disc will painlessly translate to ISO-9660:

1- The characters used in the File Names must only be A-Z, 0-9, and _. See figure 3, on page 6.

2- The depth of the directory structure can not exceed 8 levels. See figure 5, on page 9.

3- The length of the path to any file can not exceed 255 characters. See Table 1, on page 10.

To improve performance, you may also want to minimize the number of files in each directory. If there are over 50 files in a directory, you may notice some slowing while reading files in this directory, If there are over 250 files, you are very likely to notice that it takes much longer to read some files in this directory.

2. What do I need to do to make my Macintosh data ready for ISO-9660?

See question 1. In order for a disc to have Macintosh applications on it, the ISO-9660 data must include the Apple extensions, otherwise, every file on the disc will appear to be a text file to the Macintosh. Also, ISO-9660 discs will have the version number appended to each file name. See Macintosh implementation of ISO-9660, page 23, and Apple Extensions to ISO-9660, page 26.

3. Why use ISO 9660 on the Mac and what are the affects?

The primary reason for using ISO-9660 on a disc that will be running on the Macintosh is that the same disc will need to run on other platforms, such as MS-DOS. The primary disadvantage to using ISO-9660 on the Macintosh is the loss of some of the "look and feel" of the Macintosh user interface. In particular, you must live with the restrictions on the file names (see File Names, page 12), and you can not control where the folders open up to and how they are viewed (see Macintosh implementation of ISO-9660, page 23, and Apple Extensions to ISO-9660, page 26).

4. What is a hybrid disc; what are the issues to consider?

In an effort to alleviate the disadvantages to using ISO-9660 on the Macintosh (see question 3), a scheme called the hybrid disc has been developed. A hybrid disc is basically a disc with two partitions on it. It has both an ISO-9660 partition and an HFS partition. When this disc is mounted on the Macintosh, the Macintosh only sees the HFS partition. On other platforms, only the ISO-9660 partition is visible.

Until recently, the primary disadvantage to a hybrid disc was that any data that was common to the different platforms had to be duplicated in both partitions. This could significantly limit the amount of data you could place on a hybrid disc. There is now becoming available a type of hybrid disc that blends the two partitions. This allows us to only put data on the disc once, and have both ISO-9660 and HFS directory structures point to the same data.

5. How do I make one disc that works on both my PC and Macintosh?

see questions 3 and 4. Often, a disc such as this is created entirely on the Macintosh. An important point to note here is that if the ISO-9660 partition is created on the Macintosh, any files that must be readable by MS-DOS must meet ISO-9660 level 1 Interchange. That is, the file names can not be longer than 8.3 (see Levels of Interchange, page 16 and Implementations of ISO-9660, DOS, page 20).

6 .What do I need to do to make my UNIX data ready for ISO-9660?

See question 1. The primary differences between ISO-9660 for UNIX and MS-DOS is that UNIX generally can support Interchange level 2 (file names longer than 8.3), and UNIX typically can handle larger subdirectories with less performance degradation. An important point to remember when dealing with ISO-9660 and UNIX is that the file names can appear different on different systems. Even on the same system, different mount options can affect how, or if, the system translates the ISO-9660 file names. (See Implementations of ISO-9660, UNIX, page 24).

7. Will ISO discs work on UNIX, MAC and DOS?

Yes, but... Only files that meet the proper level of Interchange will be readable on any given platform, i.e., UNIX and Mac files that are recorded longer than 8.3 will not be readable by DOS (see Implementations of ISO-9660, DOS, page 19).

8. What is an Image file?

In the jargon of the industry, an Image file is a single data file that contains all of the data that will be recorded to a CD. Typically, this file will contain a complete ISO-9660 volume, or an HFS volume, or a hybrid image with both ISO and HFS volumes. If you can generate an image file, your data is *image ready.*

9. How do I create an Image file?

There are software packages available that will create an ISO-9660 volume and record it to an Image file. These packages are normally call pre-mastering packages and are available from companies such as Dataware Technologies, Meridian Data, Optical Media International, and Trace. There are too many packages to try and list them all here. The best way to choose one is to read reviews and talk to people who are using the package you are considering purchasing.

10. How do I send the Image file to the Mastering Facility?

Most pre-mastering packages contain options to write an ISO-9660 volume to a SCSI tape drive as well as to an Image file. Some packages also support writing to a CD Recordable device. Most mastering houses will accept 8mm Exabyte tapes, 4mm DDS tapes, and CD-R discs as standard input. If none of these options are available, but you do have an Image file, the Image file may be backed up with a standard backup program such as tar on UNIX systems, Retrospect on the Macintosh, and Sytos+ or Novaback on DOS. If you will be sending your Image file this way, be sure and call the mastering house to make sure they can restore the particular backup format you are using.

The JPEG Still Picture Compression Standard

Gregory K. Wallace
Multimedia Engineering
Digital Equipment Corporation
Maynard, Massachusetts

Submitted in December 1991 for publication in IEEE Transactions on Consumer Electronics

This paper is a revised version of an article by the same title and author which appeared in the April 1991 issue of Communications of the ACM.

Abstract

For the past few years, a joint ISO/CCITT committee known as JPEG (Joint Photographic Experts Group) has been working to establish the first international compression standard for continuous-tone still images, both grayscale and color. JPEG's proposed standard aims to be generic, to support a wide variety of applications for continuous-tone images. To meet the differing needs of many applications, the JPEG standard includes two basic compression methods, each with various modes of operation. A DCT-based method is specified for "lossy'' compression, and a predictive method for "lossless'' compression. JPEG features a simple lossy technique known as the Baseline method, a subset of the other DCT-based modes of operation. The Baseline method has been by far the most widely implemented JPEG method to date, and is sufficient in its own right for a large number of applications. This article provides an overview of the JPEG standard, and focuses in detail on the Baseline method.

1 Introduction

Advances over the past decade in many aspects of digital technology - especially devices for image acquisition, data storage, and bitmapped printing and display - have brought about many applications of digital imaging. However, these applications tend to be specialized due to their relatively high cost. With the possible exception of facsimile, digital images are not commonplace in general-purpose computing systems the way text and geometric graphics are. The majority of modern business and consumer usage of photographs and other types of images takes place through more traditional analog means.

The key obstacle for many applications is the vast amount of data required to represent a digital image directly. A digitized version of a single, color picture at TV resolution contains on the order of one million bytes; 35mm resolution requires ten times that amount. Use of digital images often is not viable due to high storage or transmission costs, even when image capture and display devices are quite affordable.

Modern image compression technology offers a possible solution. State-of-the-art techniques can compress typical images from 1/10 to 1/50 their uncompressed size without visibly affecting image quality. But compression technology alone is not sufficient. For digital image applications involving storage or transmission to become widespread in today's marketplace, a standard image compression method is needed to enable interoperability of equipment from different manufacturers. The CCITT recommendation for today's ubiquitous Group 3 fax machines [17] is a dramatic example of how a standard compression method can enable an important image application. The Group 3 method, however, deals with bilevel images only and does not address photographic image compression.

For the past few years, a standardization effort known by the acronym JPEG, for Joint Photographic Experts Group, has been working toward establishing the first international digital image compression standard for continuous-tone (multilevel) still images, both grayscale and color. The "joint" in JPEG refers to a collaboration between CCITT and ISO. JPEG convenes officially as the ISO committee designated JTC1/SC2/WG10, but operates in close informal collaboration with CCITT SGVIII. JPEG will be both an ISO Standard and a CCITT Recommendation. The text of both will be identical.

Photovideotex, desktop publishing, graphic arts, color facsimile, newspaper wirephoto transmission, medical imaging, and many other continuous-tone image applications require a compression standard in order to develop significantly beyond their present state. JPEG has undertaken the ambitious task of developing a general-purpose compression standard to meet the needs of almost all continuous-tone still-image applications.

If this goal proves attainable, not only will individual applications flourish, but exchange of images across application boundaries will be facilitated. This latter feature will become increasingly important as more image applications are implemented on general-purpose computing systems, which are themselves becoming increasingly interoperable and internetworked. For applications which require specialized VLSI to meet their compression and decompression speed requirements, a common method will provide economies of scale not possible within a single application.

This article gives an overview of JPEG's proposed image-compression standard. Readers without prior knowledge of JPEG or compression based on the Discrete Cosine Transform (DCT) are encouraged to study first the detailed description of the Baseline sequential codec, which is the basis for all of the DCT-based decoders. While this article provides many details, many more are necessarily omitted. The reader should refer to the ISO draft standard [2] before attempting implementation.

Some of the earliest industry attention to the JPEG proposal has been focused on the Baseline sequential codec as a motion image compression method - of the ‘‘intraframe’’ class, where each frame is encoded as a separate image. This class of motion image coding, while providing less compression than ‘‘interframe’’ methods like MPEG, has greater flexibility for video editing. While this paper focuses only on JPEG as a still picture standard (as ISO intended), it is interesting to note that JPEG is likely to become a ‘‘de facto’’ intraframe motion standard as well.

2 Background: Requirements and Selection Process

JPEG's goal has been to develop a method for continuous-tone image compression which meets the following requirements:

1) be at or near the state of the art with regard to compression rate and accompanying image fidelity, over a wide range of image quality ratings, and especially in the range where visual fidelity to the original is characterized as “very good” to “excellent”; also, the encoder should be parameterizable, so that the application (or user) can set the desired compression/quality tradeoff;

2) be applicable to practically any kind of continuous-tone digital source image (i.e. for most practical purposes not be restricted to images of certain dimensions, color spaces, pixel aspect ratios, etc.) and not be limited to classes of imagery with restrictions on scene content, such as complexity, range of colors, or statistical properties;

3) have tractable computational complexity, to make feasible software implementations with viable performance on a range of CPU's, as well as hardware implementations with viable cost for applications requiring high performance;

4) have the following modes of operation:

- Sequential encoding: each image component is encoded in a single left-to-right, top-to-bottom scan;

- Progressive encoding: the image is encoded in multiple scans for applications in which transmission time is long, and the viewer prefers to watch the image build up in multiple coarse-to-clear passes;

- Lossless encoding: the image is encoded to guarantee exact recovery of every source image sample value (even though the result is low compression compared to the lossy modes);

- Hierarchical encoding: the image is encoded at multiple resolutions so that lower-resolution versions may be accessed without first having to decompress the image at its full resolution.

In June 1987, JPEG conducted a selection process based on a blind assessment of subjective picture quality, and narrowed 12 proposed methods to three. Three informal working groups formed to refine them, and in January 1988, a second, more rigorous selection process [19] revealed that the “ADCT” proposal [11], based on the 8x8 DCT, had produced the best picture quality.

At the time of its selection, the DCT-based method was only partially defined for some of the modes of operation. From 1988 through 1990, JPEG undertook the sizable task of defining, documenting, simulating, testing, validating, and simply agreeing on the plethora of details necessary for genuine interoperability and universality. Further history of the JPEG effort is contained in [6, 7, 9, 18].

3 Architecture of the Proposed Standard

The proposed standard contains the four "modes of operation" identified previously. For each mode, one or more distinct codecs are specified. Codecs within a mode differ according to the precision of source image samples they can handle or the entropy coding method they use. Although the word codec (encoder/decoder) is used frequently in this article, there is no requirement that implementations must include both an encoder and a decoder. Many applications will have systems or devices which require only one or the other.

The four modes of operation and their various codecs have resulted from JPEG's goal of being generic and from the diversity of image formats across applications. The multiple pieces can give the impression of undesirable complexity, but they should actually be regarded as a comprehensive "toolkit" which can span a wide range of continuous-tone image applications. It is unlikely that many implementations will utilize every tool -- indeed, most of the early implementations now on the market (even before final ISO approval) have implemented only the Baseline sequential codec.

The Baseline sequential codec is inherently a rich and sophisticated compression method which will be sufficient for many applications. Getting this minimum JPEG capability implemented properly and interoperably will provide the industry with an important initial capability for exchange of images across vendors and applications.

4 Processing Steps for DCT-Based Coding

Figures 1 and 2 show the key processing steps which are the heart of the DCT-based modes of operation. These figures illustrate the special case of single-component (grayscale) image compression. The reader can grasp the essentials of DCT-based compression by thinking of it as essentially compression of a stream of 8x8 blocks of grayscale image samples. Color image compression can then be approximately regarded as compression of multiple grayscale images, which are either compressed entirely one at a time, or are compressed by alternately interleaving 8x8 sample blocks from each in turn.

For DCT sequential-mode codecs, which include the Baseline sequential codec, the simplified diagrams indicate how single-component compression works in a fairly complete way. Each 8x8 block is input, makes its way through each processing step, and yields output in compressed form into the data stream. For DCT progressive-mode codecs, an image buffer exists prior to the entropy coding step, so that an image can be stored and then parceled out in multiple scans with successively improving quality. For the hierarchical mode of operation, the steps shown are used as building blocks within a larger framework.

4.1 8x8 FDCT and IDCT

At the input to the encoder, source image samples are grouped into 8x8 blocks, shifted from unsigned integers with range $[0, 2^P - 1]$ to signed integers with range $[-2^{P-1}, 2^{P-1}-1]$, and input to the Forward DCT (FDCT). At the output from the decoder, the Inverse DCT (IDCT) outputs 8x8 sample blocks to form the reconstructed image. The following equations are the idealized mathematical definitions of the 8x8 FDCT and 8x8 IDCT:

$$F(u,v) = \frac{1}{4} C(u)C(v)\left[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y) * \cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right] \quad (1)$$

$$f(x,y) = \frac{1}{4}\left[\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v) * \cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right] \quad (2)$$

where: $C(u),\ C(v) = 1/\sqrt{2}$ for $u,v = 0$;

$C(u),\ C(v) = 1$ otherwise.

The DCT is related to the Discrete Fourier Transform (DFT). Some simple intuition for DCT-based compression can be obtained by viewing the FDCT as a harmonic analyzer and the IDCT as a harmonic synthesizer. Each 8x8 block of source image samples is effectively a 64-point discrete signal which is a function of the two spatial dimensions *x* and *y*. The FDCT takes such a signal as its input and decomposes it into 64 orthogonal basis signals. Each contains one of the 64 unique two-dimensional (2D) "spatial frequencies'' which comprise the input signal's "spectrum." The ouput of the FDCT is the set of 64 basis-signal amplitudes or "DCT coefficients" whose values are uniquely determined by the particular 64-point input signal.

The DCT coefficient values can thus be regarded as the relative amount of the 2D spatial frequencies contained in the 64-point input signal. The coefficient with zero frequency in both dimensions is called the "DC coefficient" and the remaining 63 coefficients are called the "AC coefficients.'' Because sample values

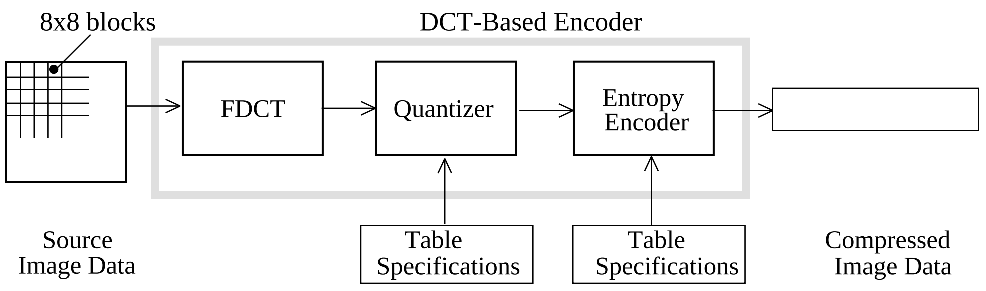


Figure 1. DCT-Based Encoder Processing Steps

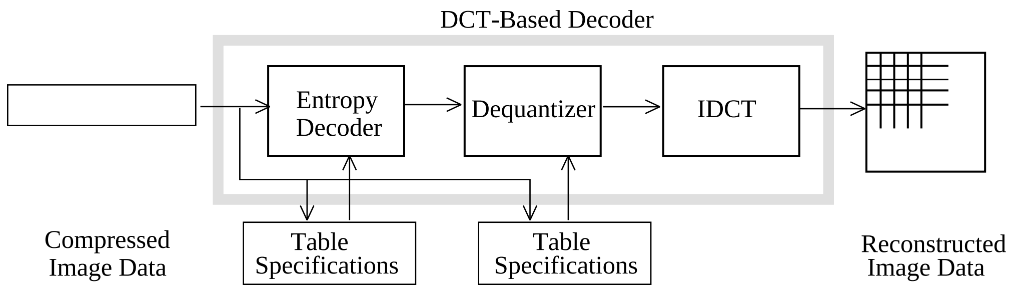


Figure 2. DCT-Based Decoder Processing Steps

typically vary slowly from point to point across an image, the FDCT processing step lays the foundation for achieving data compression by concentrating most of the signal in the lower spatial frequencies. For a typical 8x8 sample block from a typical source image, most of the spatial frequencies have zero or near-zero amplitude and need not be encoded.

At the decoder the IDCT reverses this processing step. It takes the 64 DCT coefficients (which at that point have been quantized) and reconstructs a 64-point ouput image signal by summing the basis signals. Mathematically, the DCT is one-to-one mapping for 64-point vectors between the image and the frequency domains. If the FDCT and IDCT could be computed with perfect accuracy and if the DCT coefficients were not quantized as in the following description, the original 64-point signal could be exactly recovered. In principle, the DCT introduces no loss to the source image samples; it merely transforms them to a domain in which they can be more efficiently encoded.

Some properties of practical FDCT and IDCT implementations raise the issue of what precisely should be required by the JPEG standard. A fundamental property is that the FDCT and IDCT equations contain transcendental functions. Consequently, no physical implementation can compute them with perfect accuracy. Because of the DCT's application importance and its relationship to the DFT, many different algorithms by which the FDCT and IDCT may be approximately computed have been devised [16]. Indeed, research in fast DCT algorithms is ongoing and no single algorithm is optimal for all implementations. What is optimal in software for a general-purpose CPU is unlikely to be optimal in firmware for a programmable DSP and is certain to be suboptimal for dedicated VLSI.

Even in light of the finite precision of the DCT inputs and outputs, independently designed implementations of the very same FDCT or IDCT algorithm which differ even minutely in the precision by which they represent cosine terms or intermediate results, or in the way they sum and round fractional values, will eventually produce slightly different outputs from identical inputs.

To preserve freedom for innovation and customization within implementations, JPEG has chosen to specify neither a unique FDCT algorithm or a unique IDCT algorithm in its proposed standard. This makes compliance somewhat more difficult to confirm, because two compliant encoders (or decoders) generally will not produce identical outputs given identical inputs. The JPEG standard will address this issue by specifying an accuracy test as part of its compliance tests for all DCT-based encoders and decoders; this is to ensure against crudely inaccurate cosine basis functions which would degrade image quality.

For each DCT-based mode of operation, the JPEG proposal specifies separate codecs for images with 8-bit and 12-bit (per component) source image samples. The 12-bit codecs, needed to accommodate certain types of medical and other images, require greater computational resources to achieve the required FDCT or IDCT accuracy. Images with other sample precisions can usually be accommodated by either an 8-bit or 12-bit codec, but this must be done outside the JPEG standard. For example, it would be the responsibility of an application to decide how to fit or pad a 6-bit sample into the 8-bit encoder's input interface, how to unpack it at the decoder's output, and how to encode any necessary related information.

4.2 Quantization

After output from the FDCT, each of the 64 DCT coefficients is uniformly quantized in conjunction with a 64-element Quantization Table, which must be specified by the application (or user) as an input to the encoder. Each element can be any integer value from 1 to 255, which specifies the step size of the quantizer for its corresponding DCT coefficient. The purpose of quantization is to achieve further compression by representing DCT coefficients with no greater precision than is necessary to achieve the desired image quality. Stated another way, the goal of this processing step is to discard information which is not visually significant. Quantization is a many-to-one mapping, and therefore is fundamentally lossy. It is the principal source of lossiness in DCT-based encoders.

Quantization is defined as division of each DCT coefficient by its corresponding quantizer step size, followed by rounding to the nearest integer:

$$F^Q(u,v) \;=\; \mathit{Integer\ Round}\,\left(\frac{F(u,v)}{Q(u,v)}\right) \qquad (3)$$

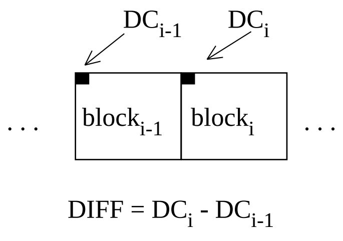
This output value is normalized by the quantizer step size. Dequantization is the inverse function, which in this case means simply that the normalization is removed by multiplying by the step size, which returns the result to a representation appropriate for input to the IDCT:

$$F^{Q'}(u,v) \;=\; F^Q(u,v) \;*\; Q(u,v) \qquad (4)$$

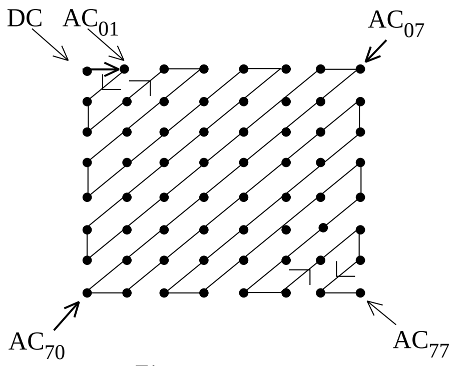
When the aim is to compress the image as much as possible without visible artifacts, each step size ideally should be chosen as the perceptual threshold or "just noticeable difference" for the visual contribution of its corresponding cosine basis function. These thresholds are also functions of the source image characteristics, display characteristics and viewing distance. For applications in which these variables can be reasonably well defined, psychovisual experiments can be performed to determine the best thresholds. The experiment described in [12] has led to a set of Quantization Tables for CCIR-601 [4] images and displays. These have been used experimentally by JPEG members and will appear in the ISO standard as a matter of information, but not as a requirement.

4.3 DC Coding and Zig-Zag Sequence

After quantization, the DC coefficient is treated separately from the 63 AC coefficients. The DC coefficient is a measure of the average value of the 64 image samples. Because there is usually strong correlation between the DC coefficients of adjacent 8x8 blocks, the quantized DC coefficient is encoded as the difference from the DC term of the previous block in the encoding order (defined in the following), as shown in Figure 3. This special treatment is worthwhile, as DC coefficients frequently contain a significant fraction of the total image energy.



Differential DC encoding



Zig−zag sequence

Figure 3. Preparation of Quantized Coefficients for Entropy Coding

Finally, all of the quantized coefficients are ordered into the “zig-zag” sequence, also shown in Figure 3. This ordering helps to facilitate entropy coding by placing low-frequency coefficients (which are more likely to be nonzero) before high-frequency coefficients.

4.4 Entropy Coding

The final DCT-based encoder processing step is entropy coding. This step achieves additional compression losslessly by encoding the quantized DCT coefficients more compactly based on their statistical characteristics. The JPEG proposal specifies two entropy coding methods - Huffman coding [8] and arithmetic coding [15]. The Baseline sequential codec uses Huffman coding, but codecs with both methods are specified for all modes of operation.

It is useful to consider entropy coding as a 2-step process. The first step converts the zig-zag sequence of quantized coefficients into an intermediate sequence of symbols. The second step converts the symbols to a data stream in which the symbols no longer have externally identifiable boundaries. The form and definition of the intermediate symbols is dependent on both the DCT-based mode of operation and the entropy coding method.

Huffman coding requires that one or more sets of Huffman code tables be specified by the application. The same tables used to compress an image are needed to decompress it. Huffman tables may be predefined and used within an application as defaults, or computed specifically for a given image in an initial statistics-gathering pass prior to compression. Such choices are the business of the applications which use JPEG; the JPEG proposal specifies no required Huffman tables. Huffman coding for the Baseline sequential encoder is described in detail in section 7.

By contrast, the particular arithmetic coding method specified in the JPEG proposal [2] requires no tables to be externally input, because it is able to adapt to the image statistics as it encodes the image. (If desired, statistical conditioning tables can be used as inputs for slightly better efficiency, but this is not required.) Arithmetic coding has produced 5-10% better compression than Huffman for many of the images which JPEG members have tested. However, some feel it is more complex than Huffman coding for certain implementations, for example, the highest-speed hardware implementations. (Throughout JPEG’s history, “complexity” has proved to be most elusive as a practical metric for comparing compression methods.)

If the only difference between two JPEG codecs is the entropy coding method, transcoding between the two is possible by simply entropy decoding with one method and entropy recoding with the other.

4.5 Compression and Picture Quality

For color images with moderately complex scenes, all DCT-based modes of operation typically produce the following levels of picture quality for the indicated ranges of compression. These levels are only a guideline - quality and compression can vary significantly according to source image characteristics and scene content. (The units “bits/pixel” here mean the total number of bits in the compressed image - including the chrominance components - divided by the number of samples in the luminance component.)

- 0.25-0.5 bits/pixel: moderate to good quality, sufficient for some applications;
- 0.5-0.75 bits/pixel: good to very good quality, sufficient for many applications;
- 0.75-1/5 bits/pixel: excellent quality, sufficient for most applications;
- 1.5-2.0 bits/pixel: usually indistinguishable from the original, sufficient for the most demanding applications.

5 Processing Steps for Predictive Lossless Coding

After its selection of a DCT-based method in 1988, JPEG discovered that a DCT-based lossless mode was difficult to define as a practical standard against which encoders and decoders could be independently implemented, without placing severe constraints on both encoder and decoder implementations.

JPEG, to meet its requirement for a lossless mode of operation, has chosen a simple predictive method which is wholly independent of the DCT processing described previously. Selection of this method was not the result of rigorous competitive evaluation as was the DCT-based method. Nevertheless, the JPEG lossless method produces results which, in light of its simplicity, are surprisingly close to the state of the art for lossless continuous-tone compression, as indicated by a recent technical report [5].

Figure 4 shows the main processing steps for a single-component image. A predictor combines the values of up to three neighboring samples (A, B, and C) to form a prediction of the sample indicated by X in Figure 5. This prediction is then subtracted from the actual value of sample X, and the difference is encoded

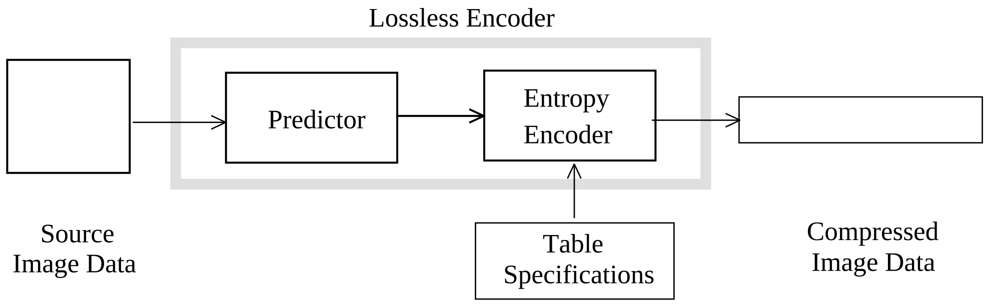


Figure 4. Lossless Mode Encoder Processing Steps

losslessly by either of the entropy coding methods - Huffman or arithmetic. Any one of the eight predictors listed in Table 1 (under "selection-value") can be used.

Selections 1, 2, and 3 are one-dimensional predictors and selections 4, 5, 6 and 7 are two-dimensional predictors. Selection-value 0 can only be used for differential coding in the hierarchical mode of operation. The entropy coding is nearly identical to that used for the DC coefficient as described in section 7.1 (for Huffman coding).

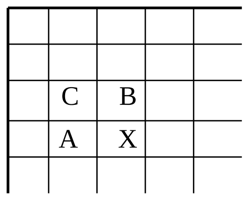


Figure 5. 3-Sample Prediction Neighborhood

For the lossless mode of operation, two different codecs are specified - one for each entropy coding method. The encoders can use any source image precision from 2 to 16 bits/sample, and can use any of the predictors except selection-value 0. The decoders must handle any of the sample precisions and any of the predictors. Lossless codecs typically produce around 2:1 compression for color images with moderately complex scenes.

selection-value	prediction
0	no prediction
1	A
2	B
3	C
4	A+B-C
5	A+((B-C)/2)
6	B+((A-C)/2)
7	(A+B)/2

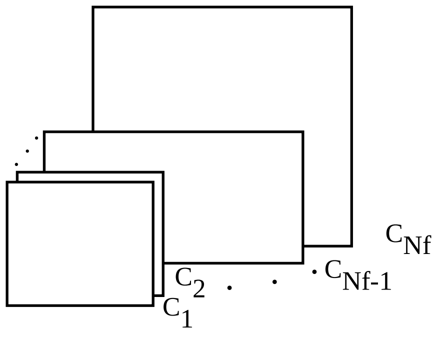
Table 1. Predictors for Lossless Coding

6 Multiple-Component Images

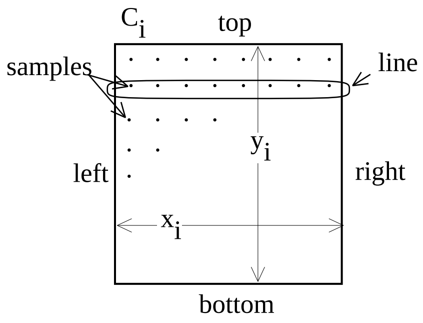
The previous sections discussed the key processing steps of the DCT-based and predictive lossless codecs for the case of single-component source images. These steps accomplish the image data compression. But a good deal of the JPEG proposal is also concerned with the handling and control of color (or other) images with multiple components. JPEG's aim for a generic compression standard requires its proposal to accommodate a variety of source image formats.

6.1 Source Image Formats

The source image model used in the JPEG proposal is an abstraction from a variety of image types and applications and consists of only what is necessary to compress and reconstruct digital image data. The reader should recognize that the JPEG compressed data format does not encode enough information to serve as a complete image representation. For example, JPEG does not specify or encode any information on pixel aspect ratio, color space, or image acquisition characteristics.



(a) Source image with multiple components



(b) Characteristics of an image component

Figure 6. JPEG Source Image Model

Figure 6 illustrates the JPEG source image model. A source image contains from 1 to 255 image components, sometimes called color or spectral bands or channels. Each component consists of a rectangular array of samples. A sample is defined to be an unsigned integer with precision P bits, with any value in the range $[0, 2^P-1]$. All samples of all components within the same source image must have the same precision P. P can be 8 or 12 for DCT-based codecs, and 2 to 16 for predictive codecs.

The ith component has sample dimensions x_i by y_i. To accommodate formats in which some image components are sampled at different rates than others, components can have different dimensions. The dimensions must have a mutual integral relationship defined by H_i and V_i, the relative horizontal and vertical sampling factors, which must be specified for each component. Overall image dimensions X and Y are defined as the maximum x_i and y_i for all components in the image, and can be any number up to 2^{16}. H and V are allowed only the integer values 1 through 4. The encoded parameters are X, Y, and H_is and V_is for each components. The decoder reconstructs the dimensions x_i and y_i for each component, according to the following relationship shown in Equation 5:

$$x_i = \left\lceil X \times \frac{H_i}{H_{max}} \right\rceil \quad \text{and}$$

$$y_i = \left\lceil Y \times \frac{V_i}{V_{max}} \right\rceil \qquad (5)$$

where $\lceil\ \rceil$ is the ceiling function.

6.2 Encoding Order and Interleaving

A practical image compression standard must address how systems will need to handle the data during the process of decompression. Many applications need to pipeline the process of displaying or printing multiple-component images in parallel with the process of decompression. For many systems, this is only feasible if the components are interleaved together within the compressed data stream.

To make the same interleaving machinery applicable to both DCT-based and predictive codecs, the JPEG proposal has defined the concept of “data unit.” A data unit is a sample in predictive codecs and an 8x8 block of samples in DCT-based codecs.

The order in which compressed data units are placed in the compressed data stream is a generalization of raster-scan order. Generally, data units are ordered from left-to-right and top-to-bottom according to the orientation shown in Figure 6. (It is the responsibility of applications to define which edges of a source image are top, bottom, left and right.) If an image component is noninterleaved (i.e., compressed without being interleaved with other components), compressed data units are ordered in a pure raster scan as shown in Figure 7.

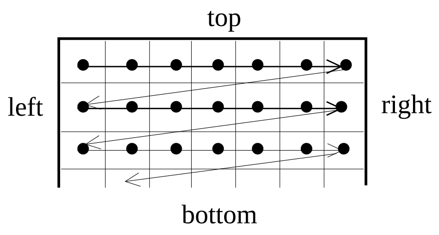
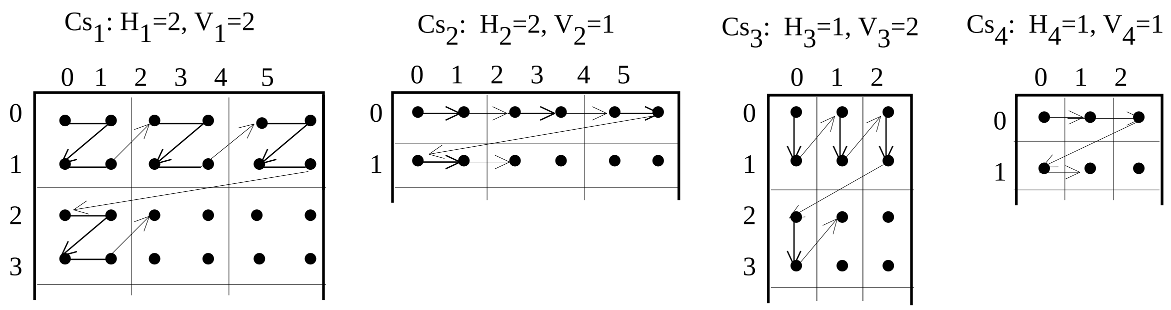


Figure 7. Noninterleaved Data Ordering

When two or more components are interleaved, each component C_i is partitioned into rectangular regions of H_i by V_i data units, as shown in the generalized example of Figure 8. Regions are ordered within a component from left-to-right and top-to-bottom, and within a region, data units are ordered from left-to-right and top-to-bottom. The JPEG proposal defines the term Minimum Coded Unit (MCU) to be the smallest



$$
\begin{aligned}
\mathrm{MCU}_1 &= d^1_{00}\, d^1_{01}\, d^1_{10}\, d^1_{11}\;\; d^2_{00}\, d^2_{01}\;\; d^3_{00}\, d^3_{10}\;\; d^4_{00},\\
\mathrm{MCU}_2 &= d^1_{02}\, d^1_{03}\, d^1_{12}\, d^1_{13}\;\; d^2_{02}\, d^2_{03}\;\; d^3_{01}\, d^3_{11}\;\; d^4_{01},\\
\mathrm{MCU}_3 &= d^1_{04}\, d^1_{05}\, d^1_{14}\, d^1_{15}\;\; d^2_{04}\, d^2_{05}\;\; d^3_{02}\, d^3_{12}\;\; d^4_{02},\\
\mathrm{MCU}_4 &= \underbrace{d^1_{20}\, d^1_{21}\, d^1_{30}\, d^1_{31}}_{\mathrm{Cs}_1 \text{ data units}}\;\; \underbrace{d^2_{10}\, d^2_{11}}_{\mathrm{Cs}_2}\;\; \underbrace{d^3_{20}\, d^3_{30}}_{\mathrm{Cs}_3}\;\; \underbrace{d^4_{10}}_{\mathrm{Cs}_4},
\end{aligned}
$$

Figure 8. Generalized Interleaved Data Ordering Example

group of interleaved data units. For the example shown, MCU_1 consists of data units taken first from the top-left-most region of C_1, followed by data units from the same region of C_2, and likewise for C_3 and C_4. MCU_2 continues the pattern as shown.

Thus, interleaved data is an ordered sequence of MCUs, and the number of data units contained in an MCU is determined by the number of components interleaved and their relative sampling factors. The maximum number of components which can be interleaved is 4 and the maximum number of data units in an MCU is 10. The latter restriction is expressed as shown in Equation 6, where the summation is over the interleaved components:

$$\sum_{\substack{\textit{all}\text{ i in}\\ \text{interleave}}} H_i \times V_i \le 10 \qquad (6)$$

Because of this restriction, not every combination of 4 components which can be represented in noninterleaved order within a JPEG-compressed image is allowed to be interleaved. Also, note that the JPEG proposal allows some components to be interleaved and some to be noninterleaved within the same compressed image.

6.3 Multiple Tables

In addition to the interleaving control discussed previously, JPEG codecs must control application of the proper table data to the proper components. The same quantization table and the same entropy coding table (or set of tables) must be used to encode all samples within a component.

JPEG decoders can store up to 4 different quantization tables and up to 4 different (sets of) entropy coding tables simultaneously. (The Baseline sequential decoder is the exception; it can only store up to 2 sets of entropy coding tables.) This is necessary for switching between different tables during decompression of a scan containing multiple (interleaved) components, in order to apply the proper table to the proper component. (Tables cannot be loaded during decompression of a scan.) Figure 9 illustrates the table-switching control that must be managed in conjunction with multiple-component interleaving for the encoder side. (This simplified view does not distinguish between quantization and entropy coding tables.)

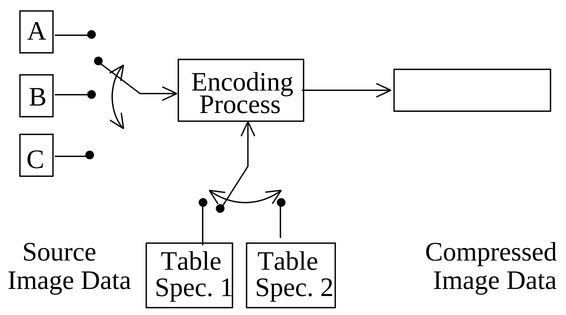


Figure 9. Component-Interleave and Table-Switching Control

7 Baseline and Other DCT Sequential Codecs

The DCT sequential mode of operation consists of the FDCT and Quantization steps from section 4, and the multiple-component control from section 6.3. In addition to the Baseline sequential codec, other DCT sequential codecs are defined to accommodate the two different sample precisions (8 and 12 bits) and the two different types of entropy coding methods (Huffman and arithmetic).

Baseline sequential coding is for images with 8-bit samples and uses Huffman coding only. It also differs from the other sequential DCT codecs in that its decoder can store only two sets of Huffman tables (one AC table and DC table per set). This restriction means that, for images with three or four interleaved components, at least one set of Huffman tables must be shared by two components. This restriction poses no limitation at all for noninterleaved components; a new set of tables can be loaded into the decoder before decompression of a noninterleaved component begins.

For many applications which do need to interleave three color components, this restriction is hardly a limitation at all. Color spaces (YUV, CIELUV, CIELAB, and others) which represent the chromatic (‘‘color’’) information in two components and the achromatic (‘‘grayscale’’) information in a third are more efficient for compression than spaces like RGB. One Huffman table set can be used for the achromatic component and one for the chrominance components. DCT coefficient statistics are similar for the chrominance components of most images, and one set of Huffman tables can encode both almost as optimally as two.

The committee also felt that early availability of single-chip implementations at commodity prices would encourage early acceptance of the JPEG proposal in a variety of applications. In 1988 when Baseline sequential was defined, the committee’s VLSI experts felt that current technology made the feasibility of crowding four sets of loadable Huffman tables - in addition to four sets of Quantization tables - onto a single commodity-priced codec chip a risky proposition.

The FDCT, Quantization, DC differencing, and zig-zag ordering processing steps for the Baseline sequential codec proceed just as described in section 4. Prior to entropy coding, there usually are few nonzero and many zero-valued coefficients. The task of entropy coding is to encode these few coefficients efficiently. The description of Baseline sequential entropy coding is given in two steps: conversion of the quantized DCT coefficients into an intermediate sequence of symbols and assignment of variable-length codes to the symbols.

7.1 Intermediate Entropy Coding Representations

In the intermediate symbol sequence, each nonzero AC coefficient is represented in combination with the ‘‘runlength’’ (consecutive number) of zero-valued AC coefficients which precede it in the zig-zag sequence. Each such runlength/nonzero-coefficient combination is (usually) represented by a pair of symbols:

symbol-1	symbol-2
(RUNLENGTH, SIZE)	(AMPLITUDE)

Symbol-1 represents two pieces of information, RUNLENGTH and SIZE. Symbol-2 represents the single piece of information designated AMPLITUDE, which is simply the amplitude of the nonzero AC coefficient. RUNLENGTH is the number of consecutive zero-valued AC coefficients in the zig-zag sequence preceding the nonzero AC coefficient being represented. SIZE is the number of bits used to encode AMPLITUDE - that is, to encoded symbol-2, by the signed-integer encoding used with JPEG’s particular method of Huffman coding.

RUNLENGTH represents zero-runs of length 0 to 15. Actual zero-runs in the zig-zag sequence can be greater than 15, so the symbol-1 value (15, 0) is interpreted as the extension symbol with runlength=16. There can be up to three consecutive (15, 0) extensions before the terminating symbol-1 whose RUNLENGTH value completes the actual runlength. The terminating symbol-1 is always followed by a single symbol-2, except for the case in which the last run of zeros includes the last (63d) AC coefficient. In this frequent case, the special symbol-1 value (0,0) means EOB (end of block), and can be viewed as an ‘‘escape’’ symbol which terminates the 8x8 sample block.

Thus, for each 8x8 block of samples, the zig-zag sequence of 63 quantized AC coefficients is represented as a sequence of symbol-1, symbol-2 symbol pairs, though each ‘‘pair’’ can have repetitions of symbol-1 in the case of a long run-length or only one symbol-1 in the case of an EOB.

The possible range of quantized AC coefficients determines the range of values which both the AMPLITUDE and the SIZE information must represent. A numerical analysis of the 8x8 FDCT equation shows that, if the 64-point (8x8 block) input signal contains N-bit integers, then the nonfractional part of the output numbers (DCT coefficients) can grow by at most 3 bits. This is also the largest possible size of a quantized DCT coefficient when its quantizer step size has integer value 1.

Baseline sequential has 8-bit integer source samples in the range [-2^7, 2^7-1], so quantized AC coefficient amplitudes are covered by integers in the range [-2^{10}, $2^{10}-1$]. The signed-integer encoding uses symbol-2 AMPLITUDE codes of 1 to 10 bits in length (so SIZE also represents values from 1 to 10), and RUNLENGTH represents values from 0 to 15 as discussed previously. For AC coefficients, the structure of the symbol-1 and symbol-2 intermediate representations is illustrated in Tables 2 and 3, respectively.

The intermediate representation for an 8x8 sample block's differential DC coefficient is structured similarly. Symbol-1, however, represents only SIZE information; symbol-2 represents AMPLITUDE information as before:

symbol-1	symbol-2
(SIZE)	(AMPLITUDE)

Because the DC coefficient is differentially encoded, it is covered by twice as many integer values, [-2^{11}, $2^{11}-1$] as the AC coefficients, so one additional level must be added to the bottom of Table 3 for DC coefficients. Symbol-1 for DC coefficients thus represents a value from 1 to 11.

		SIZE					
		0	1	2	. . .	9	10
RUN LENGTH	0	EOB					
	.	X					
	.	X		RUN-SIZE values			
	.	X					
	15	ZRL					

Table 2. Baseline Huffman Coding Symbol-1 Structure

7.2 Variable-Length Entropy Coding

Once the quantized coefficient data for an 8x8 block is represented in the intermediate symbol sequence described above, variable-length codes are assigned. For each 8x8 block, the DC coefficient's symbol-1 and symbol-2 representation is coded and output first.

For both DC and AC coefficients, each symbol-1 is encoded with a variable-length code (VLC) from the Huffman table set assigned to the 8x8 block's image component. Each symbol-2 is encoded with a "variable-length integer" (VLI) code whose length in bits is given in Table 3. VLCs and VLIs both are codes with variable lengths, but VLIs are not Huffman codes. An important distinction is that the length of a VLC (Huffman code) is not known until it is decoded, but the length of a VLI is stored in its preceding VLC.

Huffman codes (VLCs) must be specified externally as an input to JPEG encoders. (Note that the form in which Huffman tables are represented in the data stream is an indirect specification with which the decoder must construct the tables themselves prior to decompression.) The JPEG proposal includes an example set of Huffman tables in its information annex, but because they are application-specific, it specifies none for required use. The VLI codes in contrast, are "hardwired" into the proposal. This is appropriate, because the VLI codes are far more numerous, can be computed rather than stored, and have not been shown to be appreciably more efficient when implemented as Huffman codes.

7.3 Baseline Encoding Example

This section gives an example of Baseline compression and encoding of a single 8x8 sample block. Note that a good deal of the operation of a complete JPEG Baseline encoder is omitted here, including creation of Interchange Format information (parameters, headers, quantization and Huffman tables), byte-stuffing, padding to byte-boundaries prior to a marker code, and other key operations. Nonetheless, this example should help to make concrete much of the foregoing explanation.

Figure 10(a) is an 8x8 block of 8-bit samples, aribtrarily extracted from a real image. The small variations from sample to sample indicate the predominance of low spatial frequencies. After subtracting 128 from each sample for the required level-shift, the 8x8 block is input to the FDCT, equation (1). Figure 10(b) shows (to one decimal place) the resulting DCT coefficients. Except for a few of the lowest frequency coefficients, the amplitudes are quite small.

139	144	149	153	155	155	155	155
144	151	153	156	159	156	156	156
150	155	160	163	158	156	156	156
159	161	162	160	160	159	159	159
159	160	161	162	162	155	155	155
161	161	161	161	160	157	157	157
162	162	161	163	162	157	157	157
162	162	161	161	163	158	158	158

(a) source image samples

235.6	-1.0	-12.1	-5.2	2.1	-1.7	-2.7	1.3
-22.6	-17.5	-6.2	-3.2	-2.9	-0.1	0.4	-1.2
-10.9	-9.3	-1.6	1.5	0.2	-0.9	-0.6	-0.1
-7.1	-1.9	0.2	1.5	0.9	-0.1	0.0	0.3
-0.6	-0.8	1.5	1.6	-0.1	-0.7	0.6	1.3
1.8	-0.2	1.6	-0.3	-0.8	1.5	1.0	-1.0
-1.3	-0.4	-0.3	-1.5	-0.5	1.7	1.1	-0.8
-2.6	1.6	-3.8	-1.8	1.9	1.2	-0.6	-0.4

(b) forward DCT coefficients

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

(c) quantization table

15	0	-1	0	0	0	0	0
-2	-1	0	0	0	0	0	0
-1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(d) normalized quantized coefficients

240	0	-10	0	0	0	0	0
-24	-12	0	0	0	0	0	0
-14	-13	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(e) denormalized quantized coefficients

144	146	149	152	154	156	156	156
148	150	152	154	156	156	156	156
155	156	157	158	158	157	156	155
160	161	161	162	161	159	157	155
163	163	164	163	162	160	158	156
163	164	164	164	162	160	158	157
160	161	162	162	162	161	159	158
158	159	161	161	162	161	159	158

(f) reconstructed image samples

Figure 10. DCT and Quantization Examples

Figure 10(c) is the example quantization table for luminance (grayscale) components included in the informational annex of the draft JPEG standard part 1 [2]. Figure 10(d) shows the quantized DCT coefficients, normalized by their quantization table entries, as specified by equation (3). At the decoder these numbers are “denormalized” according to equation (4), and input to the IDCT, equation (2). Finally, figure 10(f) shows the reconstructed sample values, remarkably similar to the originals in 10(a).

Of course, the numbers in figure 10(d) must be Huffman-encoded before transmission to the decoder. The first number of the block to be encoded is the DC term, which must be differentially encoded. If the quantized DC term of the previous block is, for example, 12, then the difference is +3. Thus, the intermediate representation is (2)(3), for SIZE=2 and AMPLITUDE=3.

Next, the the quantized AC coefficients are encoded. Following the zig-zag order, the first non-zero coefficient is -2, preceded by a zero-run of 1. This yields an intermediate representation of (1,2)(-2). Next encountered in the zig-zag order are three consecutive non-zeros of amplitude -1. This means each is preceded by a zero-run of length zero, for intermediate symbols (0,1)(-1). The last non-zero coefficient is -1 preceded by two zeros, for (2,1)(-1). Because this is the last non-zero coefficient, the final symbol representing this 8x8 block is EOB, or (0,0).

Thus, the intermediate sequence of symbols for this example 8x8 block is:

(2)(3), (1,2)(-2), (0,1)(-1), (0,1)(-1),
(0,1)(-1), (2,1)(-1), (0,0)

Next the codes themselves must be assigned. For this example, the VLCs (Huffman codes) from the informational annex of [2] will be used. The differential-DC VLC for this example is:

(2)	011

The AC luminance VLCs for this example are:

(0,0)	1010
(0,1)	00
(1,2)	11011
(2,1)	11100

The VLIs specified in [2] are related to the two's complement representation. They are:

(3)	11
(-2)	01
(-1)	0

Thus, the bit-stream for this 8x8 example block is as follows. Note that 31 bits are required to represent 64 coefficients, which achieves compression of just under 0.5 bits/sample:

0111111011010000000001110001010

7.4 Other DCT Sequential Codecs

The structure of the 12-bit DCT sequential codec with Huffman coding is a straightforward extension of the entropy coding method described previously. Quantized DCT coefficients can be 4 bits larger, so the SIZE and AMPLITUDE information extend accordingly. DCT sequential with arithmetic coding is described in detail in [2].

8 DCT Progressive Mode

The DCT progressive mode of operation consists of the same FDCT and Quantization steps (from section 4) that are used by DCT sequential mode. The key difference is that each image component is encoded in multiple scans rather than in a single scan. The first scan(s) encode a rough but recognizable version of the image which can be transmitted quickly in comparison to the total transmission time, and are refined by succeeding scans until reaching a level of picture quality that was established by the quantization tables.

To achieve this requires the addition of an image-sized buffer memory at the output of the quantizer, before the input to entropy encoder. The buffer memory must be of sufficient size to store the image as quantized DCT coefficients, each of which (if stored straightforwardly) is 3 bits larger than the source image samples. After each block of DCT coefficients is quantized, it is stored in the coefficient buffer memory. The buffered coefficients are then partially encoded in each of multiple scans.

There are two complementary methods by which a block of quantized DCT coefficients may be partially encoded. First, only a specified "band" of coefficients from the zig-zag sequence need be encoded within a given scan. This procedure is called "spectral selection," because each band typically contains coefficients which occupy a lower or higher part of the spatial-frequency spectrum for that 8x8 block. Secondly, the coefficients within the current band need not be encoded to their full (quantized) accuracy in a given scan. Upon a coefficient's first encoding, the N most significant bits can be encoded first, where N is specifiable. In subsequent scans, the less significant bits can then be encoded. This procedure is called ''successive approximation.'' Both procedures can be used separately, or mixed in flexible combinations.

SIZE	AMPLITUDE
1	-1,1
2	-3,-2,2,3
3	-7..-4,4..7
4	-15..-8,8..15
5	-31..-16,16..31
6	-63..-32,32..63
7	-127..-64,64..127
8	-255..-128,128..255
9	-511..-256,256..511
10	-1023..-512,512..1023

Table 3. Baseline Entropy Coding Symbol-2 Structure

Some intuition for spectral selection and successive approximation can be obtained from Figure 11. The quantized DCT coefficient information can be viewed as a rectangle for which the axes are the DCT coefficients (in zig-zag order) and their amplitudes. Spectral selection slices the information in one dimension and successive approximation in the other.

9 Hierarchical Mode of Operation

The hierarchical mode provides a "pyramidal" encoding of an image at multiple resolutions, each differing in resolution from its adjacent encoding by a factor of two in either the horizontal or vertical dimension or both. The encoding procedure can be summarized as follows:

1) Filter and down-sample the original image by the desired number of multiples of 2 in each dimension.

2) Encode this reduced-size image using one of the sequential DCT, progressive DCT, or lossless encoders described previously.

3) Decode this reduced-size image and then interpolate and up-sample it by 2 horizontally and/or vertically, using the identical interpolation filter which the receiver must use.

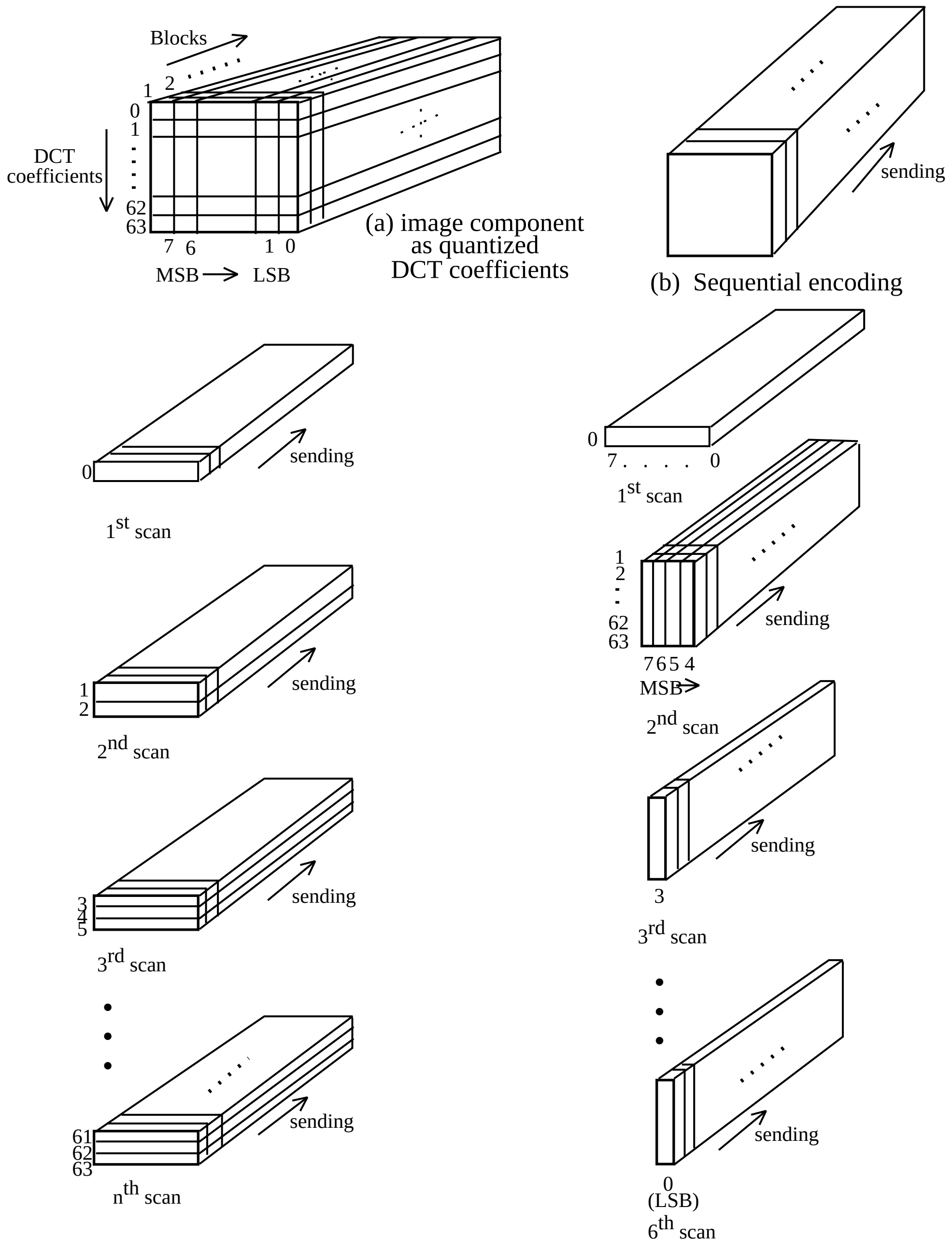


Figure 11. Spectral Selection and Successive Approximation Methods of Progressive Encoding

4) Use this up-sampled image as a prediction of the original at this resolution, and encode the difference image using one of the sequential DCT, progressive DCT, or lossless encoders described previously.

5) Repeat steps 3) and 4) until the full resolution of the image has been encoded.

The encoding in steps 2) and 4) must be done using only DCT-based processes, only lossless processes, or DCT-based processes with a final lossless process for each component.

Hierarchical encoding is useful in applications in which a very high resolution image must be accessed by a lower-resolution display. An example is an image scanned and compressed at high resolution for a very high-quality printer, where the image must also be displayed on a low-resolution PC video screen.

10 Other Aspects of the JPEG Proposal

Some key aspects of the proposed standard can only be mentioned briefly. Foremost among these are points concerning the coded representation for compressed image data specified in addition to the encoding and decoding procedures.

Most importantly, an *interchange format* syntax is specified which ensures that a JPEG-compressed image can be exchanged successfully between different application environments. The format is structured in a consistent way for all modes of operation. The interchange format always includes all quantization and entropy-coding tables which were used to compress the image.

Applications (and application-specific standards) are the “users” of the JPEG standard. The JPEG standard imposes no requirement that, within an application’s environment, all or even any tables must be encoded with the compressed image data during storage or transmission. This leaves applications the freedom to specify default or referenced tables if they are considered appropriate. It also leaves them the responsibility to ensure that JPEG-compliant decoders used within their environment get loaded with the proper tables at the proper times, and that the proper tables are included in the interchange format when a compressed image is “exported” outside the application.

Some of the important applications that are already in the process of adopting JPEG compression or have stated their interest in doing so are Adobe’s PostScript language for printing systems [1], the Raster Content portion of the ISO Office Document Architecture and Interchange Format [13], the future CCITT color facsimile standard, and the European ETSI videotext standard [10].

11 Standardization Schedule

JPEG’s ISO standard will be divided into two parts. Part 1 [2] will specify the four modes of operation, the different codecs specified for those modes, and the interchange format. It will also contain a substantial informational section on implementation guidelines. Part 2 [3] will specify the compliance tests which will determine whether an encoder implementation, a decoder implementation, or a JPEG-compressed image in interchange format comply with the Part 1 specifications. In addition to the ISO documents referenced, the JPEG standard will also be issued as CCITT Recommendation T.81.

There are two key balloting phases in the ISO standardization process: a Committee Draft (CD) is balloted to determine promotion to Draft International Standard (DIS), and a DIS is balloted to determine promotion to International Standard (IS). A CD ballot requires four to six months of processing, and a DIS ballot requires six to nine months of processing. JPEG’s Part 1 began DIS ballot in November 1991, and Part 2 began CD ballot in December 1991.

Though there is no guarantee that the first ballot of each phase will result in promotion to the next, JPEG achieved promotion of CD Part 1 to DIS Part 1 in the first ballot. Moreover, JPEG’s DIS Part 1 has undergone no technical changes (other than some minor corrections) since JPEG’s final Working Draft (WD) [14]. Thus, Part 1 has remained unchanged from the final WD, through CD, and into DIS. If all goes well, Part 1 should receive final approval as an IS in mid-1992, with Part 2 getting final IS approval about nine months later.

12 Conclusions

The emerging JPEG continuous-tone image compression standard is not a panacea that will solve the myriad issues which must be addressed before digital images will be fully integrated within all the applications that will ultimately benefit from them. For example, if two applications cannot exchange uncompressed images because they use incompatible color spaces, aspect ratios, dimensions, etc. then a common compression method will not help.

However, a great many applications are “stuck” because of storage or transmission costs, because of argument over which (nonstandard) compression method to use, or because VLSI codecs are too expensive due to low volumes. For these applications, the thorough technical evaluation, testing, selection, validation, and documentation work which JPEG committee members have performed is expected to soon yield an approved international standard that will withstand the tests of quality and time. As diverse imaging applications become increasingly implemented on open networked computing systems, the ultimate measure of the committee’s success will be when JPEG-compressed digital images come to be regarded and even taken for granted as “just another data type,” as text and graphics are today.

For more information

Information on how to obtain the ISO JPEG (draft) standards can be obtained by writing the author at the following address:

Digital Equipment Corporation
146 Main Street, ML01-2/U44
Maynard, MA 01754-2571

Internet: wallace@gauss.enet.dec.com

Floppy disks containing uncompressed, compressed, and reconstructed data for the purpose of informally validating whether an encoder or decoder implementation conforms to the proposed standard are available. Thanks to the following JPEG committee member and his company who have agreed to provide these for a nominal fee on behalf of the committee until arrangements can be made for ISO to provide them:

Eric Hamilton
C-Cube Microsystems
1778 McCarthy Blvd.
Milpitas, CA 95035

Acknowledgments

The following longtime JPEG core members have spent untold hours (usually in addition to their ‘‘real jobs’’) to make this collaborative international effort succeed. Each has made specific substantive contributions to the JPEG proposal: Aharon Gill (Zoran, Israel), Eric Hamilton (C-Cube, USA), Alain Leger (CCETT, France), Adriaan Ligtenberg (Storm, USA), Herbert Lohscheller (ANT, Germany), Joan Mitchell (IBM, USA), Michael Nier (Kodak, USA), Takao Omachi (NEC, Japan), William Pennebaker (IBM, USA), Henning Poulsen (KTAS, Denmark), and Jorgen Vaaben (AutoGraph, Denmark). The leadership efforts of Hiroshi Yasuda (NTT, Japan), the Convenor of JTC1/SC2/WG8 from which JPEG was spawned, Istvan Sebestyen (Siemens, Germany), the Special Rapporteur from CCITT SGVIII, and Graham Hudson (British Telecom U.K.) former JPEG chair and founder of the effort which became JPEG. The author regrets that space does not permit recognition of the many other individuals who contributed to JPEG’s work.

Thanks to Majid Rabbani of Eastman Kodak for providing the example in section 7.3.

The author’s role within JPEG has been supported in a great number of ways by Digital Equipment Corporation

References

1. Adobe Systems Inc. *PostScript Language Reference Manual*. Second Ed. Addison Wesley, Menlo Park, Calif. 1990
2. Digital Compression and Coding of Continuous-tone Still Images, Part 1, Requirements and Guidelines. ISO/IEC JTC1 Draft International Standard 10918-1, Nov. 1991.
3. Digital Compression and Coding of Continuous-tone Still Images, Part 2, Compliance Testing. ISO/IEC JTC1 Committee Draft 10918-2, Dec. 1991.
4. Encoding parameters of digital television for studios. CCIR Recommendations, Recommendation 601, 1982.
5. Howard, P.G., and Vitter, J.S. New methods for lossless image compression using arithmetic coding. Brown University Dept. of Computer Science Tech. Report No. CS-91-47, Aug. 1991.
6. Hudson, G.P. The development of photographic videotex in the UK. In *Proceedings of the IEEE Global Telecommunications Conference, IEEE Communication Society,* 1983, pp. 319-322.
7. Hudson, G.P., Yasuda, H., and Sebestyén, I. The international standardization of a still picture compression technique. In *Proceedings of the IEEE Global Telecommunications Conference, IEEE Communications Society,* Nov. 1988, pp. 1016-1021.
8. Huffman, D.A. A method for the construction of minimum redundancy codes. In *Proceedings IRE*, vol. 40, 1962, pp. 1098-1101.
9. Léger, A. Implementations of fast discrete cosine transform for full color videotex services and terminals. In *Proceedings of the IEEE Global Telecommunications Conference, IEEE Communications Society,* 1984, pp. 333-337.
10. Léger, A., Omachi, T., and Wallace, G. The JPEG still picture compression algorithm. In *Optical Engineering*, vol. 30, no. 7 (July 1991), pp. 947-954.
11. Léger, A., Mitchell, M., and Yamazaki, Y. Still picture compression algorithms evaluated for international standardization. In P*roceedings of the IEEE Global Telecommunications Conference, IEEE Communications Society,* Nov. 1988, pp. 1028-1032.
12. Lohscheller, H. A subjectively adapted image communication system. *IEEE Trans. Commun.* COM-32 (Dec. 1984), pp. 1316-1322.

13. Office Document Architecture (ODA) and Interchange Format, Part 7: Raster Graphics Content Architectures. ISO/IEC JTC1 International Standard 8613-7.

14. Pennebaker, W.B., JPEG Tech. Specification, Revision 8. Informal Working paper JPEG-8-R8, Aug. 1990.

15. Pennebaker, W.B., Mitchell, J.L., et. al. Arithmetic coding articles. *IBM J. Res. Dev.*, vol. 32, no. 6 (Nov. 1988), pp. 717-774.

16. Rao, K.R., and Yip, P. *Discrete Cosine Transform--Algorithms, Advantages, Applications*. Academic Press, Inc. London, 1990.

17. Standardization of Group 3 facsimile apparatus for document transmission. CCITT Recommendations, Fascicle VII.2, Recommendation T.4, 1980.

18. Wallace, G.K. Overview of the JPEG (ISO/CCITT) still image compression standard. Image Processing Algorithms and Techniques. In *Proceedings of the SPIE*, vol. 1244 (Feb. 1990), pp. 220-233.

19. Wallace, G., Vivian, R,. and Poulsen, H. Subjective testing results for still picture compression algorithms for international standardization. In *Proceedings of the IEEE Global Telecommunications Conference. IEEE Communications Society*, Nov. 1988, pp. 1022-1027.

Biography

Gregory K. Wallace is currently Manager of Multimedia Engineering, Advanced Development, at Digital Equipment Corporation. Since 1988 he has served as Chair of the JPEG committee (ISO/IEC JTC1/SC2/WG10). For the past five years at DEC, he has worked on efficient software and hardware implementations of image compression and processing algorithms for incorporation in general-purpose computing systems. He received the BSEE and MSEE from Stanford University in 1977 and 1979. His current research interests are the integration of robust real-time multimedia capabilities into networked computing systems.

INTERNATIONAL TELECOMMUNICATION UNION

CCITT

T.81

(09/92)

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

TERMINAL EQUIPMENT AND PROTOCOLS FOR TELEMATIC SERVICES

INFORMATION TECHNOLOGY – DIGITAL COMPRESSION AND CODING OF CONTINUOUS-TONE STILL IMAGES – REQUIREMENTS AND GUIDELINES

Recommendation T.81

Foreword

ITU (International Telecommunication Union) is the United Nations Specialized Agency in the field of telecommunications. The CCITT (the International Telegraph and Telephone Consultative Committee) is a permanent organ of the ITU. Some 166 member countries, 68 telecom operating entities, 163 scientific and industrial organizations and 39 international organizations participate in CCITT which is the body which sets world telecommunications standards (Recommendations).

The approval of Recommendations by the members of CCITT is covered by the procedure laid down in CCITT Resolution No. 2 (Melbourne, 1988). In addition, the Plenary Assembly of CCITT, which meets every four years, approves Recommendations submitted to it and establishes the study programme for the following period.

In some areas of information technology, which fall within CCITT's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC. The text of CCITT Recommendation T.81 was approved on 18th September 1992. The identical text is also published as ISO/IEC International Standard 10918-1.

CCITT NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized private operating agency.

Contents

Introduction

This CCITT Recommendation | ISO/IEC International Standard was prepared by CCITT Study Group VIII and the Joint Photographic Experts Group (JPEG) of ISO/IEC JTC 1/SC 29/WG 10. This Experts Group was formed in 1986 to establish a standard for the sequential progressive encoding of continuous tone grayscale and colour images.

Digital Compression and Coding of Continuous-tone Still images, is published in two parts:

– Requirements and guidelines;

– Compliance testing.

This part, Part 1, sets out requirements and implementation guidelines for continuous-tone still image encoding and decoding processes, and for the coded representation of compressed image data for interchange between applications. These processes and representations are intended to be generic, that is, to be applicable to a broad range of applications for colour and grayscale still images within communications and computer systems. Part 2, sets out tests for determining whether implementations comply with the requirments for the various encoding and decoding processes specified in Part 1.

The user's attention is called to the possibility that – for some of the coding processes specified herein – compliance with this Recommendation | International Standard may require use of an invention covered by patent rights. See Annex L for further information.

The requirements which these processes must satisfy to be useful for specific image communications applications such as facsimile, Videotex and audiographic conferencing are defined in CCITT Recommendation T.80. The intent is that the generic processes of Recommendation T.80 will be incorporated into the various CCITT Recommendations for terminal equipment for these applications.

In addition to the applications addressed by the CCITT and ISO/IEC, the JPEG committee has developped a compression standard to meet the needs of other applications as well, including desktop publishing, graphic arts, medical imaging and scientific imaging.

Annexes A, B, C, D, E, F, G, H and J are normative, and thus form an integral part of this Specification. Annexes K, L and M are informative and thus do not form an integral part of this Specification.

This Specification aims to follow the guidelines of CCITT and ISO/IEC JTC 1 on *Rules for presentation of CCITT | ISO/IEC common text*.

INTERNATIONAL STANDARD

CCITT RECOMMENDATION

INFORMATION TECHNOLOGY – DIGITAL COMPRESSION AND CODING OF CONTINUOUS-TONE STILL IMAGES – REQUIREMENTS AND GUIDELINES

1 Scope

This CCITT Recommendation | International Standard is applicable to continuous-tone – grayscale or colour – digital still image data. It is applicable to a wide range of applications which require use of compressed images. It is not applicable to bi-level image data.

This Specification

– specifies processes for converting source image data to compressed image data;

– specifies processes for converting compressed image data to reconstructed image data;

– gives guidance on how to implement these processes in practice;

– specifies coded representations for compressed image data.

NOTE – This Specification does not specify a complete coded image representation. Such representations may include certain parameters, such as aspect ratio, component sample registration, and colour space designation, which are application-dependent.

2 Normative references

The following CCITT Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this CCITT Recommendation | International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this CCITT Recommendation | International Standard are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The CCITT Secretariat maintains a list of currently valid CCITT Recommendations.

– *CCITT Recommendation T.80 (1992), Common components for image compression and communication – Basic principles.*

3 Definitions, abbreviations and symbols

3.1 Definitions and abbreviations

For the purposes of this Specification, the following definitions apply.

3.1.1 abbreviated format: A representation of compressed image data which is missing some or all of the table specifications required for decoding, or a representation of table-specification data without frame headers, scan headers, and entropy-coded segments.

3.1.2 AC coefficient: Any DCT coefficient for which the frequency is not zero in at least one dimension.

3.1.3 (adaptive) (binary) arithmetic decoding: An entropy decoding procedure which recovers the sequence of symbols from the sequence of bits produced by the arithmetic encoder.

3.1.4 (adaptive) (binary) arithmetic encoding: An entropy encoding procedure which codes by means of a recursive subdivision of the probability of the sequence of symbols coded up to that point.

3.1.5 application environment: The standards for data representation, communication, or storage which have been established for a particular application.

3.1.6 **arithmetic decoder:** An embodiment of arithmetic decoding procedure.

3.1.7 **arithmetic encoder:** An embodiment of arithmetic encoding procedure.

3.1.8 **baseline (sequential):** A particular sequential DCT-based encoding and decoding process specified in this Specification, and which is required for all DCT-based decoding processes.

3.1.9 **binary decision:** Choice between two alternatives.

3.1.10 **bit stream:** Partially encoded or decoded sequence of bits comprising an entropy-coded segment.

3.1.11 **block:** An 8×8 array of samples or an 8×8 array of DCT coefficient values of one component.

3.1.12 **block-row:** A sequence of eight contiguous component lines which are partitioned into 8×8 blocks.

3.1.13 **byte:** A group of 8 bits.

3.1.14 **byte stuffing:** A procedure in which either the Huffman coder or the arithmetic coder inserts a zero byte into the entropy-coded segment following the generation of an encoded hexadecimal X'FF' byte.

3.1.15 **carry bit:** A bit in the arithmetic encoder code register which is set if a carry-over in the code register overflows the eight bits reserved for the output byte.

3.1.16 **ceiling function:** The mathematical procedure in which the greatest integer value of a real number is obtained by selecting the smallest integer value which is greater than or equal to the real number.

3.1.17 **class (of coding process):** Lossy or lossless coding processes.

3.1.18 **code register:** The arithmetic encoder register containing the least significant bits of the partially completed entropy-coded segment. Alternatively, the arithmetic decoder register containing the most significant bits of a partially decoded entropy-coded segment.

3.1.19 **coder:** An embodiment of a coding process.

3.1.20 **coding:** Encoding or decoding.

3.1.21 **coding model:** A procedure used to convert input data into symbols to be coded.

3.1.22 **(coding) process:** A general term for referring to an encoding process, a decoding process, or both.

3.1.23 **colour image:** A continuous-tone image that has more than one component.

3.1.24 **columns:** Samples per line in a component.

3.1.25 **component:** One of the two-dimensional arrays which comprise an image.

3.1.26 **compressed data:** Either compressed image data or table specification data or both.

3.1.27 **compressed image data:** A coded representation of an image, as specified in this Specification.

3.1.28 **compression:** Reduction in the number of bits used to represent source image data.

3.1.29 **conditional exchange:** The interchange of MPS and LPS probability intervals whenever the size of the LPS interval is greater than the size of the MPS interval (in arithmetic coding).

3.1.30 **(conditional) probability estimate:** The probability value assigned to the LPS by the probability estimation state machine (in arithmetic coding).

3.1.31 **conditioning table:** The set of parameters which select one of the defined relationships between prior coding decisions and the conditional probability estimates used in arithmetic coding.

3.1.32 **context:** The set of previously coded binary decisions which is used to create the index to the probability estimation state machine (in arithmetic coding).

3.1.33 **continuous-tone image:** An image whose components have more than one bit per sample.

3.1.34 **data unit:** An 8×8 block of samples of one component in DCT-based processes; a sample in lossless processes.

3.1.35 DC coefficient: The DCT coefficient for which the frequency is zero in both dimensions.

3.1.36 DC prediction: The procedure used by DCT-based encoders whereby the quantized DC coefficient from the previously encoded 8×8 block of the same component is subtracted from the current quantized DC coefficient.

3.1.37 (DCT) coefficient: The amplitude of a specific cosine basis function – may refer to an original DCT coefficient, to a quantized DCT coefficient, or to a dequantized DCT coefficient.

3.1.38 decoder: An embodiment of a decoding process.

3.1.39 decoding process: A process which takes as its input compressed image data and outputs a continuous-tone image.

3.1.40 default conditioning: The values defined for the arithmetic coding conditioning tables at the beginning of coding of an image.

3.1.41 dequantization: The inverse procedure to quantization by which the decoder recovers a representation of the DCT coefficients.

3.1.42 differential component: The difference between an input component derived from the source image and the corresponding reference component derived from the preceding frame for that component (in hierarchical mode coding).

3.1.43 differential frame: A frame in a hierarchical process in which differential components are either encoded or decoded.

3.1.44 (digital) reconstructed image (data): A continuous-tone image which is the output of any decoder defined in this Specification.

3.1.45 (digital) source image (data): A continuous-tone image used as input to any encoder defined in this Specification.

3.1.46 (digital) (still) image: A set of two-dimensional arrays of integer data.

3.1.47 discrete cosine transform; DCT: Either the forward discrete cosine transform or the inverse discrete cosine transform.

3.1.48 downsampling (filter): A procedure by which the spatial resolution of an image is reduced (in hierarchical mode coding).

3.1.49 encoder: An embodiment of an encoding process.

3.1.50 encoding process: A process which takes as its input a continuous-tone image and outputs compressed image data.

3.1.51 entropy-coded (data) segment: An independently decodable sequence of entropy encoded bytes of compressed image data.

3.1.52 (entropy-coded segment) pointer: The variable which points to the most recently placed (or fetched) byte in the entropy encoded segment.

3.1.53 entropy decoder: An embodiment of an entropy decoding procedure.

3.1.54 entropy decoding: A lossless procedure which recovers the sequence of symbols from the sequence of bits produced by the entropy encoder.

3.1.55 entropy encoder: An embodiment of an entropy encoding procedure.

3.1.56 entropy encoding: A lossless procedure which converts a sequence of input symbols into a sequence of bits such that the average number of bits per symbol approaches the entropy of the input symbols.

3.1.57 extended (DCT-based) process: A descriptive term for DCT-based encoding and decoding processes in which additional capabilities are added to the baseline sequential process.

3.1.58 forward discrete cosine transform; FDCT: A mathematical transformation using cosine basis functions which converts a block of samples into a corresponding block of original DCT coefficients.

3.1.59 frame: A group of one or more scans (all using the same DCT-based or lossless process) through the data of one or more of the components in an image.

3.1.60 frame header: A marker segment that contains a start-of-frame marker and associated frame parameters that are coded at the beginning of a frame.

3.1.61 frequency: A two-dimensional index into the two-dimensional array of DCT coefficients.

3.1.62 (frequency) band: A contiguous group of coefficients from the zig-zag sequence (in progressive mode coding).

3.1.63 full progression: A process which uses both spectral selection and successive approximation (in progressive mode coding).

3.1.64 grayscale image: A continuous-tone image that has only one component.

3.1.65 hierarchical: A mode of operation for coding an image in which the first frame for a given component is followed by frames which code the differences between the source data and the reconstructed data from the previous frame for that component. Resolution changes are allowed between frames.

3.1.66 hierarchical decoder: A sequence of decoder processes in which the first frame for each component is followed by frames which decode an array of differences for each component and adds it to the reconstructed data from the preceding frame for that component.

3.1.67 hierarchical encoder: The mode of operation in which the first frame for each component is followed by frames which encode the array of differences between the source data and the reconstructed data from the preceding frame for that component.

3.1.68 horizontal sampling factor: The relative number of horizontal data units of a particular component with respect to the number of horizontal data units in the other components.

3.1.69 Huffman decoder: An embodiment of a Huffman decoding procedure.

3.1.70 Huffman decoding: An entropy decoding procedure which recovers the symbol from each variable length code produced by the Huffman encoder.

3.1.71 Huffman encoder: An embodiment of a Huffman encoding procedure.

3.1.72 Huffman encoding: An entropy encoding procedure which assigns a variable length code to each input symbol.

3.1.73 Huffman table: The set of variable length codes required in a Huffman encoder and Huffman decoder.

3.1.74 image data: Either source image data or reconstructed image data.

3.1.75 interchange format: The representation of compressed image data for exchange between application environments.

3.1.76 interleaved: The descriptive term applied to the repetitive multiplexing of small groups of data units from each component in a scan in a specific order.

3.1.77 inverse discrete cosine transform; IDCT: A mathematical transformation using cosine basis functions which converts a block of dequantized DCT coefficients into a corresponding block of samples.

3.1.78 Joint Photographic Experts Group; JPEG: The informal name of the committee which created this Specification. The “joint” comes from the CCITT and ISO/IEC collaboration.

3.1.79 latent output: Output of the arithmetic encoder which is held, pending resolution of carry-over (in arithmetic coding).

3.1.80 less probable symbol; LPS: For a binary decision, the decision value which has the smaller probability.

3.1.81 level shift: A procedure used by DCT-based encoders and decoders whereby each input sample is either converted from an unsigned representation to a two’s complement representation or from a two’s complement representation to an unsigned representation.

3.1.82 lossless: A descriptive term for encoding and decoding processes and procedures in which the output of the decoding procedure(s) is identical to the input to the encoding procedure(s).

3.1.83 lossless coding: The mode of operation which refers to any one of the coding processes defined in this Specification in which all of the procedures are lossless (see Annex H).

3.1.84 lossy: A descriptive term for encoding and decoding processes which are not lossless.

3.1.85 marker: A two-byte code in which the first byte is hexadecimal FF (X'FF') and the second byte is a value between 1 and hexadecimal FE (X'FE').

3.1.86 marker segment: A marker and associated set of parameters.

3.1.87 MCU-row: The smallest sequence of MCU which contains at least one line of samples or one block-row from every component in the scan.

3.1.88 minimum coded unit; MCU: The smallest group of data units that is coded.

3.1.89 modes (of operation): The four main categories of image coding processes defined in this Specification.

3.1.90 more probable symbol; MPS: For a binary decision, the decision value which has the larger probability.

3.1.91 non-differential frame: The first frame for any components in a hierarchical encoder or decoder. The components are encoded or decoded without subtraction from reference components. The term refers also to any frame in modes other than the hierarchical mode.

3.1.92 non-interleaved: The descriptive term applied to the data unit processing sequence when the scan has only one component.

3.1.93 parameters: Fixed length integers 4, 8 or 16 bits in length, used in the compressed data formats.

3.1.94 point transform: Scaling of a sample or DCT coefficient.

3.1.95 precision: Number of bits allocated to a particular sample or DCT coefficient.

3.1.96 predictor: A linear combination of previously reconstructed values (in lossless mode coding).

3.1.97 probability estimation state machine: An interlinked table of probability values and indices which is used to estimate the probability of the LPS (in arithmetic coding).

3.1.98 probability interval: The probability of a particular sequence of binary decisions within the ordered set of all possible sequences (in arithmetic coding).

3.1.99 (probability) sub-interval: A portion of a probability interval allocated to either of the two possible binary decision values (in arithmetic coding).

3.1.100 procedure: A set of steps which accomplishes one of the tasks which comprise an encoding or decoding process.

3.1.101 process: See coding process.

3.1.102 progressive (coding): One of the DCT-based processes defined in this Specification in which each scan typically improves the quality of the reconstructed image.

3.1.103 progressive DCT-based: The mode of operation which refers to any one of the processes defined in Annex G.

3.1.104 quantization table: The set of 64 quantization values used to quantize the DCT coefficients.

3.1.105 quantization value: An integer value used in the quantization procedure.

3.1.106 quantize: The act of performing the quantization procedure for a DCT coefficient.

3.1.107 reference (reconstructed) component: Reconstructed component data which is used in a subsequent frame of a hierarchical encoder or decoder process (in hierarchical mode coding).

3.1.108 renormalization: The doubling of the probability interval and the code register value until the probability interval exceeds a fixed minimum value (in arithmetic coding).

3.1.109 restart interval: The integer number of MCUs processed as an independent sequence within a scan.

3.1.110 restart marker: The marker that separates two restart intervals in a scan.

3.1.111 run (length): Number of consecutive symbols of the same value.

3.1.112 sample: One element in the two-dimensional array which comprises a component.

3.1.113 sample-interleaved: The descriptive term applied to the repetitive multiplexing of small groups of samples from each component in a scan in a specific order.

3.1.114 scan: A single pass through the data for one or more of the components in an image.

3.1.115 scan header: A marker segment that contains a start-of-scan marker and associated scan parameters that are coded at the beginning of a scan.

3.1.116 sequential (coding): One of the lossless or DCT-based coding processes defined in this Specification in which each component of the image is encoded within a single scan.

3.1.117 sequential DCT-based: The mode of operation which refers to any one of the processes defined in Annex F.

3.1.118 spectral selection: A progressive coding process in which the zig-zag sequence is divided into bands of one or more contiguous coefficients, and each band is coded in one scan.

3.1.119 stack counter: The count of X'FF' bytes which are held, pending resolution of carry-over in the arithmetic encoder.

3.1.120 statistical conditioning: The selection, based on prior coding decisions, of one estimate out of a set of conditional probability estimates (in arithmetic coding).

3.1.121 statistical model: The assignment of a particular conditional probability estimate to each of the binary arithmetic coding decisions.

3.1.122 statistics area: The array of statistics bins required for a coding process which uses arithmetic coding.

3.1.123 statistics bin: The storage location where an index is stored which identifies the value of the conditional probability estimate used for a particular arithmetic coding binary decision.

3.1.124 successive approximation: A progressive coding process in which the coefficients are coded with reduced precision in the first scan, and precision is increased by one bit with each succeeding scan.

3.1.125 table specification data: The coded representation from which the tables used in the encoder and decoder are generated and their destinations specified.

3.1.126 transcoder: A procedure for converting compressed image data of one encoder process to compressed image data of another encoder process.

3.1.127 (uniform) quantization: The procedure by which DCT coefficients are linearly scaled in order to achieve compression.

3.1.128 upsampling (filter): A procedure by which the spatial resolution of an image is increased (in hierarchical mode coding).

3.1.129 vertical sampling factor: The relative number of vertical data units of a particular component with respect to the number of vertical data units in the other components in the frame.

3.1.130 zero byte: The X'00' byte.

3.1.131 zig-zag sequence: A specific sequential ordering of the DCT coefficients from (approximately) lowest spatial frequency to highest.

3.1.132 3-sample predictor: A linear combination of the three nearest neighbor reconstructed samples to the left and above (in lossless mode coding).

3.2 Symbols

The symbols used in this Specification are listed below.

A	probability interval
AC	AC DCT coefficient
AC_{ji}	AC coefficient predicted from DC values
Ah	successive approximation bit position, high
Al	successive approximation bit position, low
Ap_i	*i*th 8-bit parameter in APP_n segment
APP_n	marker reserved for application segments
B	current byte in compressed data
B2	next byte in compressed data when B = X’FF’
BE	counter for buffered correction bits for Huffman coding in the successive approximation process
BITS	16-byte list containing number of Huffman codes of each length
BP	pointer to compressed data
BPST	pointer to byte before start of entropy-coded segment
BR	counter for buffered correction bits for Huffman coding in the successive approximation process
Bx	byte modified by a carry-over
C	value of bit stream in code register
C_i	component identifier for frame
C_u	horizontal frequency dependent scaling factor in DCT
C_v	vertical frequency dependent scaling factor in DCT
CE	conditional exchange
C-low	low order 16 bits of the arithmetic decoder code register
Cm_i	*i*th 8-bit parameter in COM segment
CNT	bit counter in NEXTBYTE procedure
CODE	Huffman code value
CODESIZE(V)	code size for symbol V
COM	comment marker
Cs	conditioning table value
Cs_i	component identifier for scan
CT	renormalization shift counter
Cx	high order 16 bits of arithmetic decoder code register
CX	conditional exchange
d_{ji}	data unit from horizontal position i, vertical position j
d_{ji}^k	d_{ji} for component k
D	decision decoded

Da	in DC coding, the DC difference coded for the previous block from the same component; in lossless coding, the difference coded for the sample immediately to the left
DAC	define-arithmetic-coding-conditioning marker
Db	the difference coded for the sample immediately above
DC	DC DCT coefficient
DC_i	DC coefficient for *i*th block in component
DC_k	*k*th DC value used in prediction of AC coefficients
DHP	define hierarchical progression marker
DHT	define-Huffman-tables marker
DIFF	difference between quantized DC and prediction
DNL	define-number-of-lines marker
DQT	define-quantization-tables marker
DRI	define restart interval marker
E	exponent in magnitude category upper bound
EC	event counter
ECS	entropy-coded segment
ECS_i	*i*th entropy-coded segment
Eh	horizontal expansion parameter in EXP segment
EHUFCO	Huffman code table for encoder
EHUFSI	encoder table of Huffman code sizes
EOB	end-of-block for sequential; end-of-band for progressive
EOBn	run length category for EOB runs
EOBx	position of EOB in previous successive approximation scan
EOB0, EOB1, ..., EOB14	run length categories for EOB runs
EOI	end-of-image marker
Ev	vertical expansion parameter in EXP segment
EXP	expand reference components marker
FREQ(V)	frequency of occurrence of symbol V
H_i	horizontal sampling factor for *i*th component
H_{max}	largest horizontal sampling factor
HUFFCODE	list of Huffman codes corresponding to lengths in HUFFSIZE
HUFFSIZE	list of code lengths
HUFFVAL	list of values assigned to each Huffman code
i	subscript index
I	integer variable
Index(S)	index to probability estimation state machine table for context index S
j	subscript index
J	integer variable

JPG	marker reserved for JPEG extensions
JPG_n	marker reserved for JPEG extensions
k	subscript index
K	integer variable
Kmin	index of 1st AC coefficient in band (1 for sequential DCT)
Kx	conditioning parameter for AC arithmetic coding model
L	DC and lossless coding conditioning lower bound parameter
L_i	element in BITS list in DHT segment
$L_i(t)$	element in BITS list in the DHT segment for Huffman table t
La	length of parameters in APP_n segment
LASTK	largest value of K
Lc	length of parameters in COM segment
Ld	length of parameters in DNL segment
Le	length of parameters in EXP segment
Lf	length of frame header parameters
Lh	length of parameters in DHT segment
Lp	length of parameters in DAC segment
LPS	less probable symbol (in arithmetic coding)
Lq	length of parameters in DQT segment
Lr	length of parameters in DRI segment
Ls	length of scan header parameters
LSB	least significant bit
m	modulo 8 counter for RST_m marker
m_t	number of $V_{i,j}$ parameters for Huffman table t
M	bit mask used in coding magnitude of V
Mn	*n*th statistics bin for coding magnitude bit pattern category
MAXCODE	table with maximum value of Huffman code for each code length
MCU	minimum coded unit
MCU_i	*i*th MCU
MCUR	number of MCU required to make up one MCU-row
MINCODE	table with minimum value of Huffman code for each code length
MPS	more probable symbol (in arithmetic coding)
MPS(S)	more probable symbol for context-index S
MSB	most significant bit
M2, M3, M4, ... , M15	designation of context-indices for coding of magnitude bits in the arithmetic coding models
n	integer variable
N	data unit counter for MCU coding
N/A	not applicable

Nb	number of data units in MCU
Next_Index_LPS	new value of Index(S) after a LPS renormalization
Next_Index_MPS	new value of Index(S) after a MPS renormalization
Nf	number of components in frame
NL	number of lines defined in DNL segment
Ns	number of components in scan
OTHERS(V)	index to next symbol in chain
P	sample precision
Pq	quantizer precision parameter in DQT segment
Pq(t)	quantizer precision parameter in DQT segment for quantization table t
PRED	quantized DC coefficient from the most recently coded block of the component
Pt	point transform parameter
Px	calculated value of sample
Q_{ji}	quantizer value for coefficient AC_{ji}
Q_{vu}	quantization value for DCT coefficient S_{vu}
Q_{00}	quantizer value for DC coefficient
QAC_{ji}	quantized AC coefficient predicted from DC values
QDC_k	*k*th quantized DC value used in prediction of AC coefficients
Qe	LPS probability estimate
Qe(S)	LPS probability estimate for context index S
Qk	*k*th element of 64 quantization elements in DQT segment
r_{vu}	reconstructed image sample
R	length of run of zero amplitude AC coefficients
R_{vu}	dequantized DCT coefficient
Ra	reconstructed sample value
Rb	reconstructed sample value
Rc	reconstructed sample value
Rd	rounding in prediction calculation
RES	reserved markers
Ri	restart interval in DRI segment
RRRR	4-bit value of run length of zero AC coefficients
RS	composite value used in Huffman coding of AC coefficients
RST_m	restart marker number m
s_{yx}	reconstructed value from IDCT
S	context index
S_{vu}	DCT coefficient at horizontal frequency u, vertical frequency v

SC	context-index for coding of correction bit in successive approximation coding
Se	end of spectral selection band in zig-zag sequence
SE	context-index for coding of end-of-block or end-of-band
SI	Huffman code size
SIGN	1 if decoded sense of sign is negative and 0 if decoded sense of sign is positive
SIZE	length of a Huffman code
SLL	shift left logical operation
SLL α β	logical shift left of α by β bits
SN	context-index for coding of first magnitude category when V is negative
SOF_0	baseline DCT process frame marker
SOF_1	extended sequential DCT frame marker, Huffman coding
SOF_2	progressive DCT frame marker, Huffman coding
SOF_3	lossless process frame marker, Huffman coding
SOF_5	differential sequential DCT frame marker, Huffman coding
SOF_6	differential progressive DCT frame marker, Huffman coding
SOF_7	differential lossless process frame marker, Huffman coding
SOF_9	sequential DCT frame marker, arithmetic coding
SOF_{10}	progressive DCT frame marker, arithmetic coding
SOF_{11}	lossless process frame marker, arithmetic coding
SOF_{13}	differential sequential DCT frame marker, arithmetic coding
SOF_{14}	differential progressive DCT frame marker, arithmetic coding
SOF_{15}	differential lossless process frame marker, arithmetic coding
SOI	start-of-image marker
SOS	start-of-scan marker
SP	context-index for coding of first magnitude category when V is positive
Sq_{vu}	quantized DCT coefficient
SRL	shift right logical operation
SRL α β	logical shift right of α by β bits
Ss	start of spectral selection band in zig-zag sequence
SS	context-index for coding of sign decision
SSSS	4-bit size category of DC difference or AC coefficient amplitude
ST	stack counter
Switch_MPS	parameter controlling inversion of sense of MPS
Sz	parameter used in coding magnitude of V
S0	context-index for coding of V = 0 decision
t	summation index for parameter limits computation
T	temporary variable

Ta_j	AC entropy table destination selector for *j*th component in scan
Tb	arithmetic conditioning table destination identifier
Tc	Huffman coding or arithmetic coding table class
Td_j	DC entropy table destination selector for *j*th component in scan
TEM	temporary marker
Th	Huffman table destination identifier in DHT segment
Tq	quantization table destination identifier in DQT segment
Tq_i	quantization table destination selector for *i*th component in frame
U	DC and lossless coding conditioning upper bound parameter
V	symbol or value being either encoded or decoded
V_i	vertical sampling factor for *i*th component
$V_{i,j}$	*j*th value for length *i* in HUFFVAL
V_{max}	largest vertical sampling factor
V_t	temporary variable
VALPTR	list of indices for first value in HUFFVAL for each code length
V1	symbol value
V2	symbol value
x_i	number of columns in *i*th component
X	number of samples per line in component with largest horizontal dimension
X_i	*i*th statistics bin for coding magnitude category decision
X1, X2, X3, ... , X15	designation of context-indices for coding of magnitude categories in the arithmetic coding models
XHUFCO	extended Huffman code table
XHUFSI	table of sizes of extended Huffman codes
X’values’	values within the quotes are hexadecimal
y_i	number of lines in *i*th component
Y	number of lines in component with largest vertical dimension
ZRL	value in HUFFVAL assigned to run of 16 zero coefficients
ZZ(K)	*K*th element in zig-zag sequence of quantized DCT coefficients
ZZ(0)	quantized DC coefficient in zig-zag sequence order

4 General

The purpose of this clause is to give an informative overview of the elements specified in this Specification. Another purpose is to introduce many of the terms which are defined in clause 3. These terms are printed in *italics* upon first usage in this clause.

4.1 Elements specified in this Specification

There are three elements specified in this Specification:

a) An *encoder* is an embodiment of an *encoding process*. As shown in Figure 1, an encoder takes as input *digital source image data* and *table specifications*, and by means of a specified set of *procedures* generates as output *compressed image data*.

b) A *decoder* is an embodiment of a *decoding process*. As shown in Figure 2, a decoder takes as input compressed image data and table specifications, and by means of a specified set of procedures generates as output *digital reconstructed image data.*

c) The *interchange format,* shown in Figure 3, is a compressed image data representation which includes all table specifications used in the encoding process. The interchange format is for exchange between *application environments*.

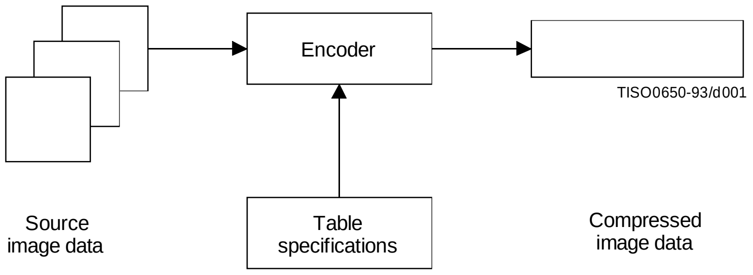


Figure 1 – Encoder

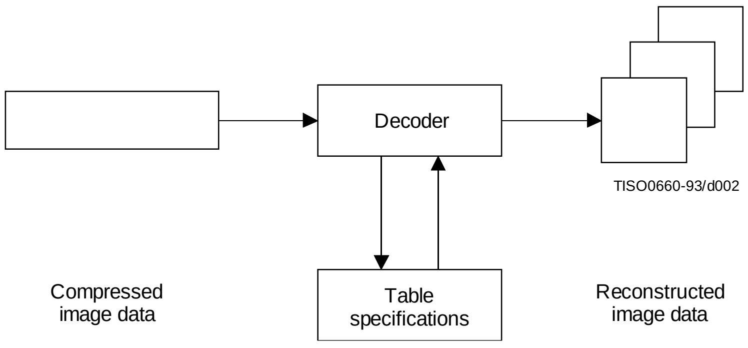


Figure 2 – Decoder

Figures 1 and 2 illustrate the general case for which the *continuous-tone* source and reconstructed image data consist of multiple *components*. (A *colour* image consists of multiple components; a *grayscale* image consists only of a single component.) A significant portion of this Specification is concerned with how to handle multiple-component images in a flexible, application-independent way.

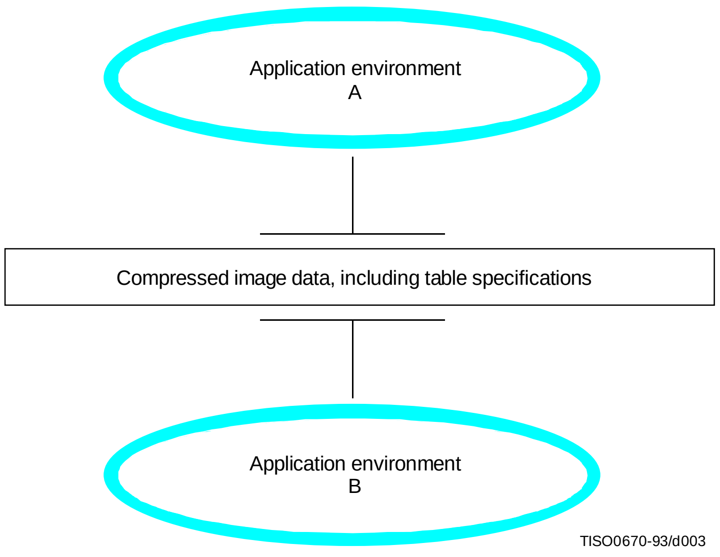


Figure 3 – Interchange format for compressed image data

These figures are also meant to show that the same tables specified for an encoder to use to compress a particular image must be provided to a decoder to reconstruct that image. However, this Specification does not specify how applications should associate tables with compressed image data, nor how they should represent source image data generally within their specific environments.

Consequently, this Specification also specifies the interchange format shown in Figure 3, in which table specifications are included within compressed image data. An image compressed with a specified encoding process within one application environment, A, is passed to a different environment, B, by means of the interchange format. The interchange format does not specify a complete coded image representation. Application-dependent information, e.g. colour space, is outside the scope of this Specification.

4.2 Lossy and lossless compression

This Specification specifies two *classes* of encoding and decoding processes, *lossy* and *lossless* processes. Those based on the *discrete cosine transform* (DCT) are lossy, thereby allowing substantial *compression* to be achieved while producing a reconstructed image with high visual fidelity to the encoder's source image.

The simplest DCT-based *coding process* is referred to as the *baseline sequential* process. It provides a capability which is sufficient for many applications. There are additional DCT-based processes which extend the baseline sequential process to a broader range of applications. In any decoder using *extended DCT-based decoding processes*, the baseline decoding process is required to be present in order to provide a default decoding capability.

The second class of coding processes is not based upon the DCT and is provided to meet the needs of applications requiring lossless compression. These lossless encoding and decoding processes are used independently of any of the DCT-based processes.

A table summarizing the relationship among these lossy and lossless coding processes is included in 4.11.

The amount of compression provided by any of the various processes is dependent on the characteristics of the particular image being compressed, as well as on the picture quality desired by the application and the desired speed of compression and decompression.

4.3 DCT-based coding

Figure 4 shows the main procedures for all encoding processes based on the DCT. It illustrates the special case of a single-component image; this is an appropriate simplification for overview purposes, because all processes specified in this Specification operate on each image component independently.

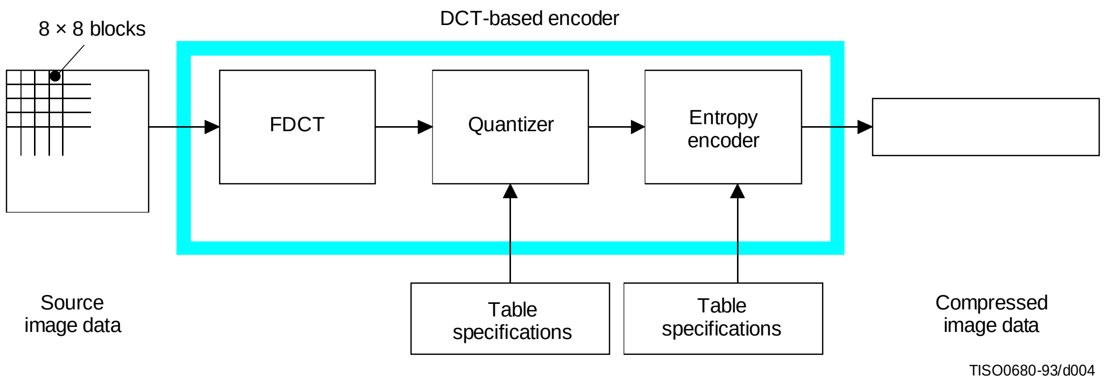


Figure 4 – DCT-based encoder simplified diagram

In the encoding process the input component's *samples* are grouped into 8 × 8 *blocks*, and each block is transformed by the *forward DCT* (FDCT) into a set of 64 values referred to as *DCT coefficients*. One of these values is referred to as the *DC coefficient* and the other 63 as the *AC coefficients*.

Each of the 64 coefficients is then *quantized* using one of 64 corresponding values from a *quantization table* (determined by one of the table specifications shown in Figure 4). No default values for quantization tables are specified in this Specification; applications may specify values which customize picture quality for their particular image characteristics, display devices, and viewing conditions.

After quantization, the DC coefficient and the 63 AC coefficients are prepared for *entropy encoding*, as shown in Figure 5. The previous quantized DC coefficient is used to predict the current quantized DC coefficient, and the difference is encoded. The 63 quantized AC coefficients undergo no such differential encoding, but are converted into a one-dimensional *zig-zag sequence*, as shown in Figure 5.

The quantized coefficients are then passed to an entropy encoding procedure which compresses the data further. One of two entropy coding procedures can be used, as described in 4.6. If *Huffman encoding* is used, *Huffman table* specifications must be provided to the encoder. If *arithmetic encoding* is used, arithmetic coding *conditioning table* specifications may be provided, otherwise the default conditioning table specifications shall be used.

Figure 6 shows the main procedures for all DCT-based decoding processes. Each step shown performs essentially the inverse of its corresponding main procedure within the encoder. The entropy decoder decodes the zig-zag sequence of quantized DCT coefficients. After *dequantization* the DCT coefficients are transformed to an 8 × 8 block of samples by the *inverse DCT* (IDCT).

4.4 Lossless coding

Figure 7 shows the main procedures for the lossless encoding processes. A *predictor* combines the reconstructed values of up to three neighbourhood samples at positions a, b, and c to form a prediction of the sample at position x as shown in Figure 8. This prediction is then subtracted from the actual value of the sample at position x, and the difference is losslessly entropy-coded by either Huffman or arithmetic coding.

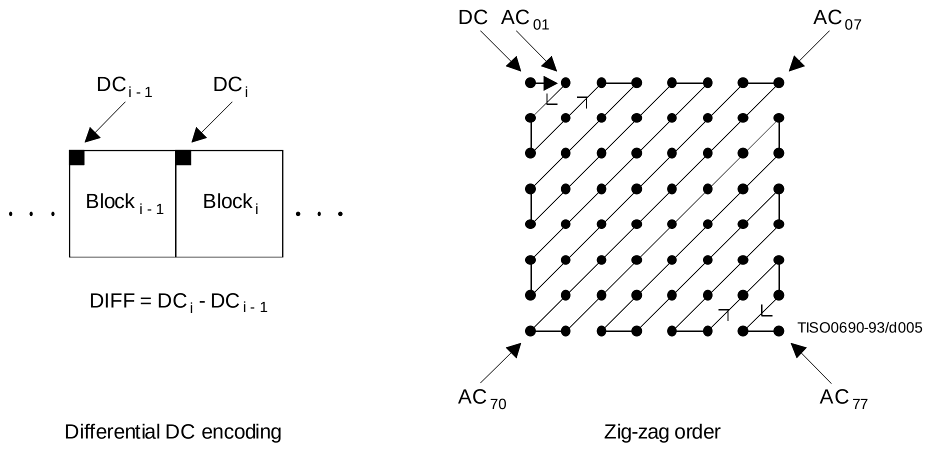


Figure 5 – Preparation of quantized coefficients for entropy encoding

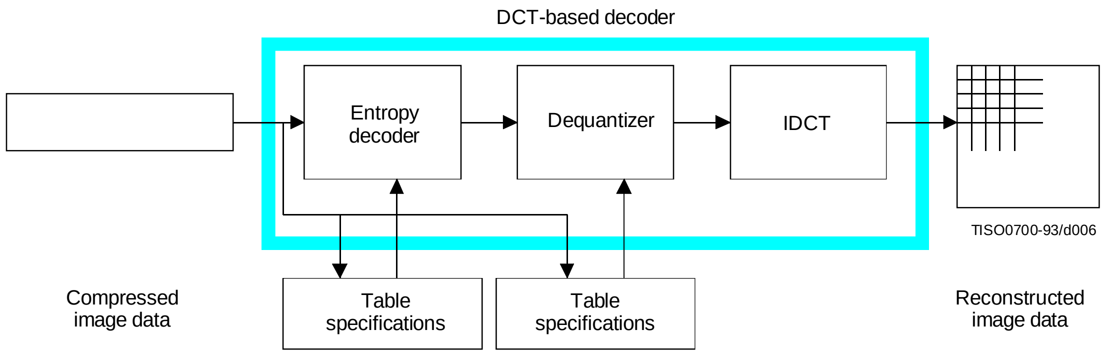


Figure 6 – DCT-based decoder simplified diagram

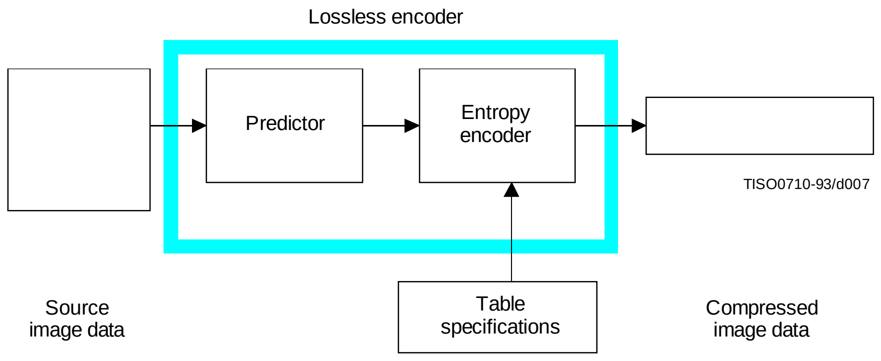


Figure 7 – Lossless encoder simplified diagram

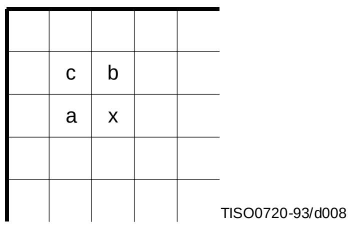


Figure 8 – 3-sample prediction neighbourhood

This encoding process may also be used in a slightly modified way, whereby the *precision* of the input samples is reduced by one or more bits prior to the lossless coding. This achieves higher compression than the lossless process (but lower compression than the DCT-based processes for equivalent visual fidelity), and limits the reconstructed image's worst-case sample error to the amount of input precision reduction.

4.5 Modes of operation

There are four distinct *modes of operation* under which the various coding processes are defined: *sequential DCT-based, progressive DCT-based,* lossless, and *hierarchical*. (Implementations are not required to provide all of these.) The lossless mode of operation was described in 4.4. The other modes of operation are compared as follows.

For the sequential DCT-based mode, 8×8 sample blocks are typically input block by block from left to right, and block-row by block-row from top to bottom. After a block has been transformed by the forward DCT, quantized and prepared for entropy encoding, all 64 of its quantized DCT coefficients can be immediately entropy encoded and output as part of the compressed image data (as was described in 4.3), thereby minimizing coefficient storage requirements.

For the progressive DCT-based mode, 8×8 blocks are also typically encoded in the same order, but in multiple *scans* through the image. This is accomplished by adding an image-sized coefficient memory buffer (not shown in Figure 4) between the quantizer and the entropy encoder. As each block is transformed by the forward DCT and quantized, its coefficients are stored in the buffer. The DCT coefficients in the buffer are then partially encoded in each of multiple scans. The typical sequence of image presentation at the output of the decoder for sequential versus progressive modes of operation is shown in Figure 9.

There are two procedures by which the quantized coefficients in the buffer may be partially encoded within a scan. First, only a specified *band* of coefficients from the zig-zag sequence need be encoded. This procedure is called *spectral selection*, because each band typically contains coefficients which occupy a lower or higher part of the *frequency* spectrum for that 8×8 block. Secondly, the coefficients within the current band need not be encoded to their full (quantized) accuracy within each scan. Upon a coefficient's first encoding, a specified number of most significant bits is encoded first. In subsequent scans, the less significant bits are then encoded. This procedure is called *successive approximation*. Either procedure may be used separately, or they may be mixed in flexible combinations.

In hierarchical mode, an image is encoded as a sequence of *frames*. These frames provide *reference reconstructed components* which are usually needed for prediction in subsequent frames. Except for the first frame for a given component, *differential frames* encode the difference between source components and reference reconstructed components. The coding of the differences may be done using only DCT-based processes, only lossless processes, or DCT-based processes with a final lossless process for each component. *Downsampling* and *upsampling filters* may be used to provide a pyramid of spatial resolutions as shown in Figure 10. Alternatively, the hierarchical mode can be used to improve the quality of the reconstructed components at a given spatial resolution.

Hierarchical mode offers a progressive presentation similar to the progressive DCT-based mode but is useful in environments which have multi-resolution requirements. Hierarchical mode also offers the capability of progressive coding to a final lossless stage.

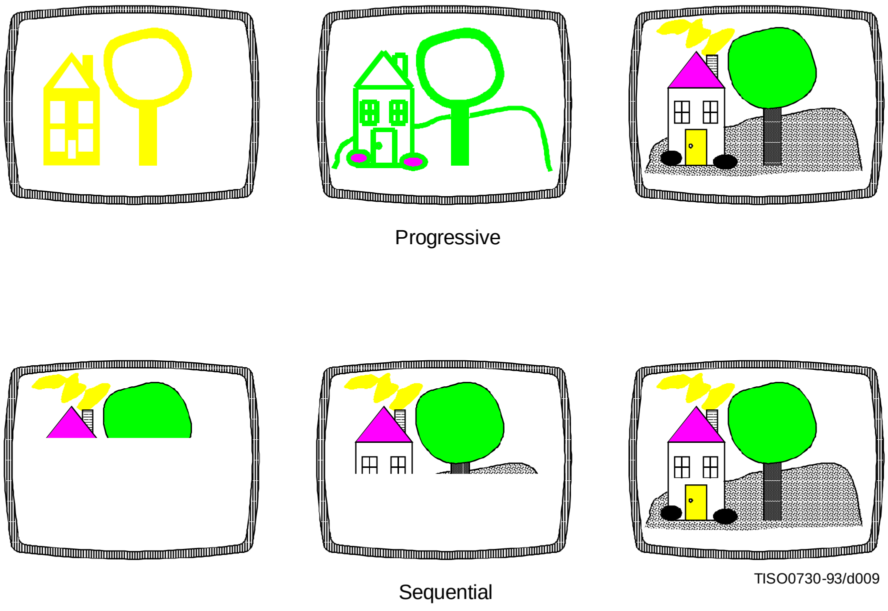


Figure 9 – Progressive versus sequential presentation

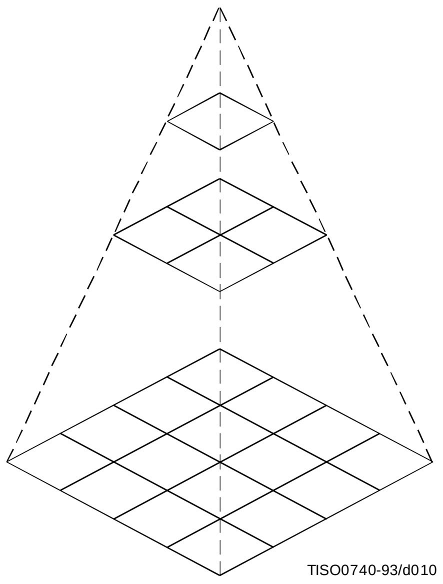


Figure 10 – Hierarchical multi-resolution encoding

4.6 Entropy coding alternatives

Two alternative entropy coding procedures are specified: Huffman coding and arithmetic coding. Huffman coding procedures use Huffman tables, determined by one of the table specifications shown in Figures 1 and 2. Arithmetic coding procedures use arithmetic coding conditioning tables, which may also be determined by a table specification. No default values for Huffman tables are specified, so that applications may choose tables appropriate for their own environments. Default tables are defined for the arithmetic coding conditioning.

The baseline sequential process uses Huffman coding, while the extended DCT-based and lossless processes may use either Huffman or arithmetic coding.

4.7 Sample precision

For DCT-based processes, two alternative sample precisions are specified: either 8 bits or 12 bits per sample. Applications which use samples with other precisions can use either 8-bit or 12-bit precision by shifting their source image samples appropriately. The baseline process uses only 8-bit precision. DCT-based implementations which handle 12-bit source image samples are likely to need greater computational resources than those which handle only 8-bit source images. Consequently in this Specification separate normative requirements are defined for 8-bit and 12-bit DCT-based processes.

For lossless processes the sample precision is specified to be from 2 to 16 bits.

4.8 Multiple-component control

Subclauses 4.3 and 4.4 give an overview of one major part of the encoding and decoding processes – those which operate on the sample values in order to achieve compression. There is another major part as well – the procedures which control the order in which the image data from multiple components are processed to create the compressed data, and which ensure that the proper set of table data is applied to the proper *data units* in the image. (A data unit is a sample for lossless processes and an 8×8 block of samples for DCT-based processes.)

4.8.1 Interleaving multiple components

Figure 11 shows an example of how an encoding process selects between multiple source image components as well as multiple sets of table data, when performing its encoding procedures. The source image in this example consists of the three components A, B and C, and there are two sets of table specifications. (This simplified view does not distinguish between the quantization tables and entropy coding tables.)

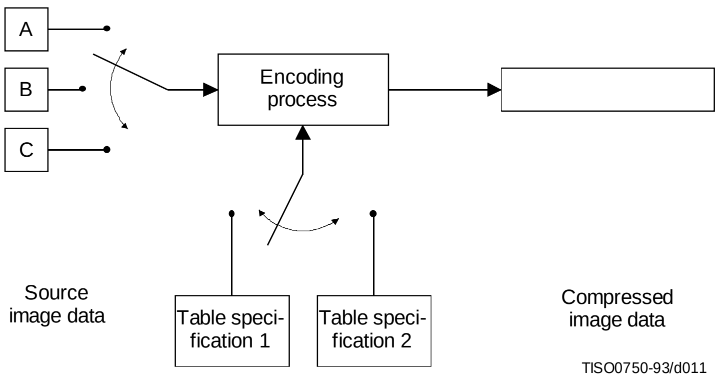


Figure 11 – Component-interleave and table-switching control

In sequential mode, encoding is *non-interleaved* if the encoder compresses all image data units in component A before beginning component B, and then in turn all of B before C. Encoding is *interleaved* if the encoder compresses a data unit from A, a data unit from B, a data unit from C, then back to A, etc. These alternatives are illustrated in Figure 12, which shows a case in which all three image components have identical dimensions: X *columns* by Y lines, for a total of n data units each.

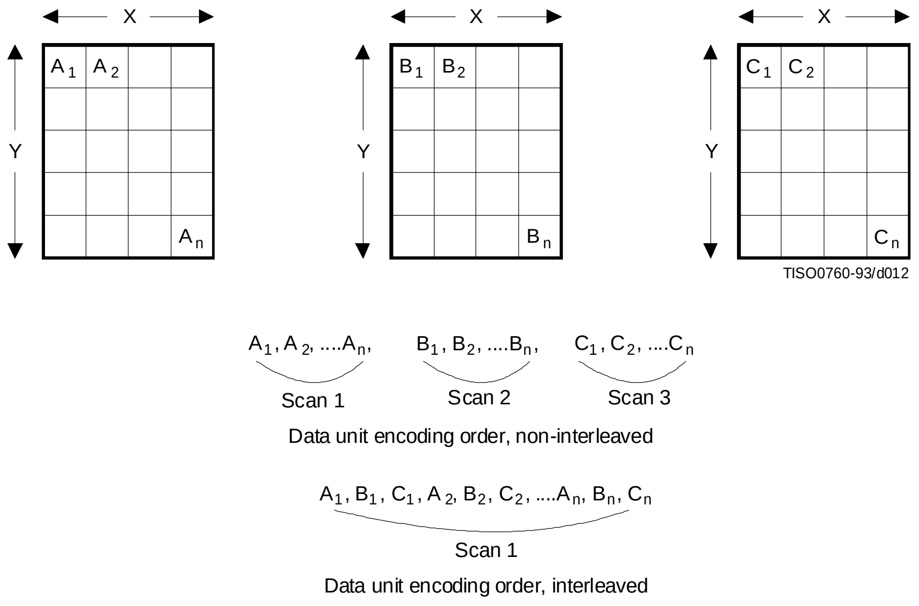

X
Y
A1 A2
An
B1 B2
Bn
C1 C2
Cn
TISO0760-93/d012
A1, A2, ...An,
B1, B2, ...Bn,
C1, C2, ...Cn
Scan 1
Scan 2
Scan 3
Data unit encoding order, non-interleaved
A1, B1, C1, A2, B2, C2, ...An, Bn, Cn
Scan 1
Data unit encoding order, interleaved

Figure 12 – Interleaved versus non-interleaved encoding order

These control procedures are also able to handle cases in which the source image components have different dimensions. Figure 13 shows a case in which two of the components, B and C, have half the number of horizontal samples relative to component A. In this case, two data units from A are interleaved with one each from B and C. Cases in which components of an image have more complex relationships, such as different horizontal and vertical dimensions, can be handled as well. (See Annex A.)

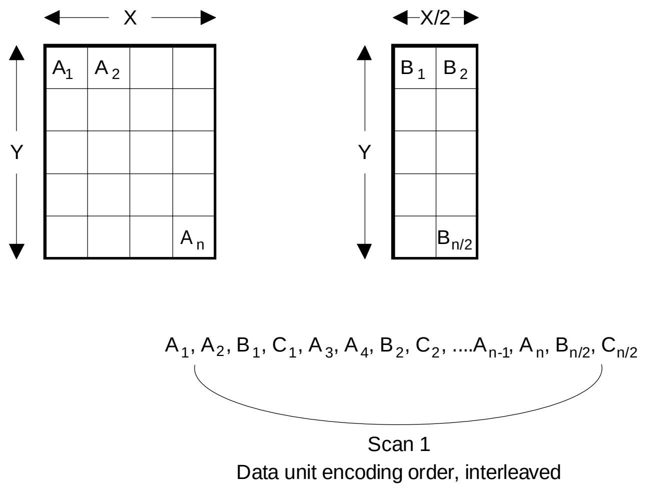

X
X/2
Y
A1 A2
An
B1 B2
Bn/2
C1 C2
Cn/2
TISO0770-93/d013
A1, A2, B1, C1, A3, A4, B2, C2, ...An-1, An, Bn/2, Cn/2
Scan 1
Data unit encoding order, interleaved

Figure 13 – Interleaved order for components with different dimensions

4.8.2 Minimum coded unit

Related to the concepts of multiple-component interleave is the *minimum coded unit* (MCU). If the compressed image data is non-interleaved, the MCU is defined to be one data unit. For example, in Figure 12 the MCU for the non-interleaved case is a single data unit. If the compressed data is interleaved, the MCU contains one or more data units from each component. For the interleaved case in Figure 12, the (first) MCU consists of the three interleaved data units A_1, B_1, C_1. In the example of Figure 13, the (first) MCU consists of the four data units A_1, A_2 , B_1, C_1.

4.9 Structure of compressed data

Figures 1, 2, and 3 all illustrate slightly different views of compressed image data. Figure 1 shows this data as the output of an encoding process, Figure 2 shows it as the input to a decoding process, and Figure 3 shows compressed image data in the interchange format, at the interface between applications.

Compressed image data are described by a uniform structure and set of *parameters* for both classes of encoding processes (lossy or lossless), and for all modes of operation (sequential, progressive, lossless, and hierarchical). The various parts of the compressed image data are identified by special two-byte codes called *markers*. Some markers are followed by particular sequences of parameters, as in the case of table specifications, *frame header*, or *scan header*. Others are used without parameters for functions such as marking the start-of-image and end-of-image. When a marker is associated with a particular sequence of parameters, the marker and its parameters comprise a *marker segment*.

The data created by the entropy encoder are also segmented, and one particular marker – *the restart marker* – is used to isolate *entropy-coded data segments*. The encoder outputs the restart markers, intermixed with the entropy-coded data, at regular *restart intervals* of the source image data. Restart markers can be identified without having to decode the compressed data to find them. Because they can be independently decoded, they have application-specific uses, such as parallel encoding or decoding, isolation of data corruptions, and semi-random access of entropy-coded segments.

There are three compressed data formats:

a) the interchange format;

b) the *abbreviated format* for compressed image data;

c) the abbreviated format for table-specification data.

4.9.1 Interchange format

In addition to certain required marker segments and the entropy-coded segments, the interchange format shall include the marker segments for all quantization and entropy-coding table specifications needed by the decoding process. This guarantees that a compressed image can cross the boundary between application environments, regardless of how each environment internally associates tables with compressed image data.

4.9.2 Abbreviated format for compressed image data

The abbreviated format for compressed image data is identical to the interchange format, except that it does not include all tables required for decoding. (It may include some of them.) This format is intended for use within applications where alternative mechanisms are available for supplying some or all of the table-specification data needed for decoding.

4.9.3 Abbreviated format for table-specification data

This format contains only table-specification data. It is a means by which the application may install in the decoder the tables required to subsequently reconstruct one or more images.

4.10 Image, frame, and scan

Compressed image data consists of only one image. An image contains only one frame in the cases of sequential and progressive coding processes; an image contains multiple frames for the hierarchical mode.

A frame contains one or more scans. For sequential processes, a scan contains a complete encoding of one or more image components. In Figures 12 and 13, the frame consists of three scans when non-interleaved, and one scan if all three components are interleaved together. The frame could also consist of two scans: one with a non-interleaved component, the other with two components interleaved.

For progressive processes, a scan contains a partial encoding of all data units from one or more image components. Components shall not be interleaved in progressive mode, except for the DC coefficients in the first scan for each component of a progressive frame.

4.11 Summary of coding processes

Table 1 provides a summary of the essential characteristics of the various coding processes specified in this Specification. The full specification of these processes is contained in Annexes F, G, H, and J.

Table 1 – Summary: Essential characteristics of coding processes

Baseline process (required for all DCT-based decoders)
• DCT-based process • Source image: 8-bit samples within each component • Sequential • Huffman coding: 2 AC and 2 DC tables • Decoders shall process scans with 1, 2, 3, and 4 components • Interleaved and non-interleaved scans

Extended DCT-based processes
• DCT-based process • Source image: 8-bit or 12-bit samples • Sequential or progressive • Huffman or arithmetic coding: 4 AC and 4 DC tables • Decoders shall process scans with 1, 2, 3, and 4 components • Interleaved and non-interleaved scans

Lossless processes
• Predictive process (not DCT-based) • Source image: P-bit samples ($2 \leq P \leq 16$) • Sequential • Huffman or arithmetic coding: 4 DC tables • Decoders shall process scans with 1, 2, 3, and 4 components • Interleaved and non-interleaved scans

Hierarchical processes
• Multiple frames (non-differential and differential) • Uses extended DCT-based or lossless processes • Decoders shall process scans with 1, 2, 3, and 4 components • Interleaved and non-interleaved scans

5 Interchange format requirements

The interchange format is the coded representation of compressed image data for exchange between application environments.

The interchange format requirements are that any compressed image data represented in interchange format shall comply with the syntax and code assignments appropriate for the decoding process selected, as specified in Annex B.

Tests for whether compressed image data comply with these requirements are specified in Part 2 of this Specification.

6 Encoder requirements

An encoding process converts source image data to compressed image data. Each of Annexes F, G, H, and J specifies a number of distinct encoding processes for its particular mode of operation.

An encoder is an embodiment of one (or more) of the encoding processes specified in Annexes F, G, H, or J. In order to comply with this Specification, an encoder shall satisfy at least one of the following two requirements.

An encoder shall

a) with appropriate accuracy, convert source image data to compressed image data which comply with the interchange format syntax specified in Annex B for the encoding process(es) embodied by the encoder;

b) with appropriate accuracy, convert source image data to compressed image data which comply with the abbreviated format for compressed image data syntax specified in Annex B for the encoding process(es) embodied by the encoder.

For each of the encoding processes specified in Annexes F, G, H, and J, the compliance tests for the above requirements are specified in Part 2 of this Specification.

NOTE – There is **no requirement** in this Specification that any encoder which embodies one of the encoding processes specified in Annexes F, G, H, or J shall be able to operate for all ranges of the parameters which are allowed for that process. An encoder is only required to meet the compliance tests specified in Part 2, and to generate the compressed data format according to Annex B for those parameter values which it does use.

7 Decoder requirements

A decoding process converts compressed image data to reconstructed image data. Each of Annexes F, G, H, and J specifies a number of distinct decoding processes for its particular mode of operation.

A decoder is an embodiment of one (or more) of the decoding processes specified in Annexes F, G, H, or J. In order to comply with this Specification, a decoder shall satisfy all three of the following requirements.

A decoder shall

a) with appropriate accuracy, convert to reconstructed image data any compressed image data with parameters within the range supported by the application, and which comply with the interchange format syntax specified in Annex B for the decoding process(es) embodied by the decoder;

b) accept and properly store any table-specification data which comply with the abbreviated format for table-specification data syntax specified in Annex B for the decoding process(es) embodied by the decoder;

c) with appropriate accuracy, convert to reconstructed image data any compressed image data which comply with the abbreviated format for compressed image data syntax specified in Annex B for the decoding process(es) embodied by the decoder, provided that the table-specification data required for decoding the compressed image data has previously been installed into the decoder.

Additionally, any DCT-based decoder, if it embodies any DCT-based decoding process other than baseline sequential, shall also embody the baseline sequential decoding process.

For each of the decoding processes specified in Annexes F, G, H, and J, the compliance tests for the above requirements are specified in Part 2 of this Specification.

Annex A

Mathematical definitions

(This annex forms an integral part of this Recommendation | International Standard)

A.1 Source image

Source images to which the encoding processes specified in this Specification can be applied are defined in this annex.

A.1.1 Dimensions and sampling factors

As shown in Figure A.1, a source image is defined to consist of Nf components. Each component, with unique identifier C_i, is defined to consist of a rectangular array of samples of x_i columns by y_i lines. The component dimensions are derived from two parameters, X and Y, where X is the maximum of the x_i values and Y is the maximum of the y_i values for all components in the frame. For each component, sampling factors H_i and V_i are defined relating component dimensions x_i and y_i to maximum dimensions X and Y, according to the following expressions:

$$x_i = \left\lceil X \times \frac{H_i}{H_{max}} \right\rceil \text{ and } y_i \left\lceil Y \times \frac{V_i}{V_{max}} \right\rceil,$$

where H_{max} and V_{max} are the maximum sampling factors for all components in the frame, and $\lceil\ \rceil$ is the ceiling function.

As an example, consider an image having 3 components with maximum dimensions of 512 lines and 512 samples per line, and with the following sampling factors:

Component 0	$H_0 = 4, \quad V_0 = 1$
Component 1	$H_1 = 2, \quad V_1 = 2$
Component 2	$H_2 = 1, \quad V_2 = 1$

Then $X = 512$, $Y = 512$, $H_{max} = 4$, $V_{max} = 2$, and x_i and y_i for each component are

Component 0	$x_0 = 512, \quad y_0 = 256$
Component 1	$x_1 = 256, \quad y_1 = 512$
Component 2	$x_2 = 128, \quad y_2 = 256$

NOTE – The X, Y, H_i, and V_i parameters are contained in the frame header of the compressed image data (see B.2.2), whereas the individual component dimensions x_i and y_i are derived by the decoder. Source images with x_i and y_i dimensions which do not satisfy the expressions above cannot be properly reconstructed.

A.1.2 Sample precision

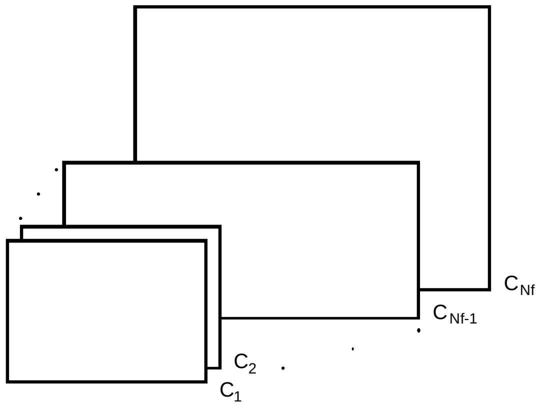
A sample is an integer with precision P bits, with any value in the range 0 through 2^{P-1}. All samples of all components within an image shall have the same precision P. Restrictions on the value of P depend on the mode of operation, as specified in B.2 to B.7.

A.1.3 Data unit

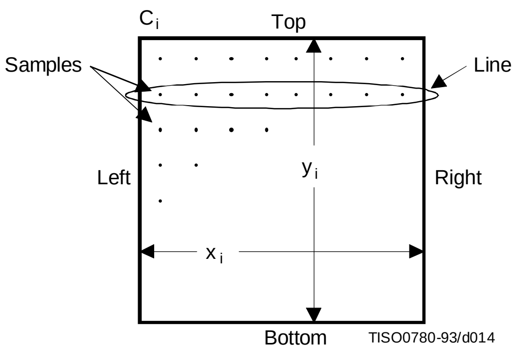
A data unit is a sample in lossless processes and an 8×8 block of contiguous samples in DCT-based processes. The left-most 8 samples of each of the top-most 8 rows in the component shall always be the top-left-most block. With this top-left-most block as the reference, the component is partitioned into contiguous data units to the right and to the bottom (as shown in Figure A.4).

A.1.4 Orientation

Figure A.1 indicates the orientation of an image component by the terms top, bottom, left, and right. The order by which the data units of an image component are input to the compression encoding procedures is defined to be left-to-right and top-to-bottom within the component. (This ordering is precisely defined in A.2.) Applications determine which edges of a source image are defined as top, bottom, left, and right.



a) Source image with multiple components



b) Characteristics of an image component

Figure A.1 – Source image characteristics

A.2 Order of source image data encoding

The scan header (see B.2.3) specifies the order by which source image data units shall be encoded and placed within the compressed image data. For a given scan, if the scan header parameter Ns = 1, then data from only one source component – the component specified by parameter Cs_1 – shall be present within the scan. This data is non-interleaved by definition. If Ns > 1, then data from the Ns components Cs_1 through Cs_{Ns} shall be present within the scan. This data shall always be interleaved. The order of components in a scan shall be according to the order specified in the frame header.

The ordering of data units and the construction of minimum coded units (MCU) is defined as follows.

A.2.1 Minimum coded unit (MCU)

For non-interleaved data the MCU is one data unit. For interleaved data the MCU is the sequence of data units defined by the sampling factors of the components in the scan.

A.2.2 Non-interleaved order (Ns = 1)

When Ns = 1 (where Ns is the number of components in a scan), the order of data units within a scan shall be left-to-right and top-to-bottom, as shown in Figure A.2. This ordering applies whenever Ns = 1, regardless of the values of H_1 and V_1.

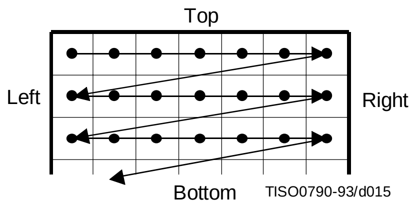
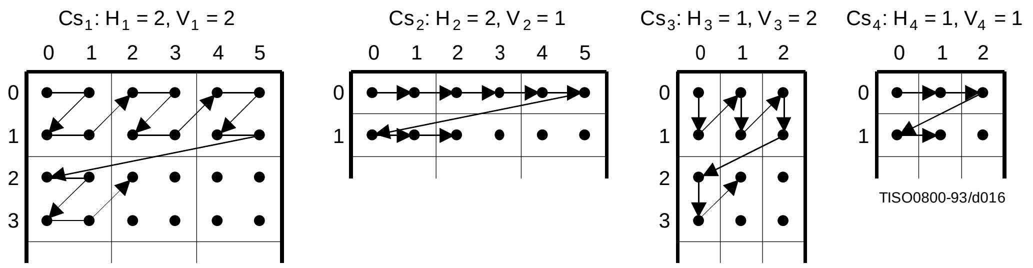


Figure A.2 – Non-interleaved data ordering

A.2.3 Interleaved order (Ns > 1)

When Ns > 1, each scan component Cs_i is partitioned into small rectangular arrays of H_k horizontal data units by V_k vertical data units. The subscripts k indicate that H_k and V_k are from the position in the frame header component-specification for which $C_k = Cs_i$. Within each H_k by V_k array, data units are ordered from left-to-right and top-to-bottom. The arrays in turn are ordered from left-to-right and top-to-bottom within each component.

As shown in the example of Figure A.3, Ns = 4, and MCU_1 consists of data units taken first from the top-left-most region of Cs_1, followed by data units from the corresponding region of Cs_2, then from Cs_3 and then from Cs_4. MCU_2 follows the same ordering for data taken from the next region to the right for the four components.



$$\begin{aligned}
MCU_1 &= d^1_{00}\ d^1_{01}\ d^1_{10}\ d^1_{11}\ \ d^2_{00}\ d^2_{01}\ \ d^3_{00}\ d^3_{10}\ \ d^4_{00},\\
MCU_2 &= d^1_{02}\ d^1_{03}\ d^1_{12}\ d^1_{13}\ \ d^2_{02}\ d^2_{03}\ \ d^3_{01}\ d^3_{11}\ \ d^4_{01},\\
MCU_3 &= d^1_{04}\ d^1_{05}\ d^1_{14}\ d^1_{15}\ \ d^2_{04}\ d^2_{05}\ \ d^3_{02}\ d^3_{12}\ \ d^4_{02},\\
MCU_4 &= \underbrace{d^1_{20}\ d^1_{21}\ d^1_{30}\ d^1_{31}}_{Cs_1 \text{ data units}}\ \ \underbrace{d^2_{10}\ d^2_{11}}_{Cs_2}\ \ \underbrace{d^3_{20}\ d^3_{30}}_{Cs_3}\ \ \underbrace{d^4_{10}}_{Cs_4},
\end{aligned}$$

Figure A.3 – Interleaved data ordering example

A.2.4 Completion of partial MCU

For DCT-based processes the data unit is a block. If x_i is not a multiple of 8, the encoding process shall extend the number of columns to complete the right-most sample blocks. If the component is to be interleaved, the encoding process shall also extend the number of samples by one or more additional blocks, if necessary, so that the number of blocks is an integer multiple of H_i. Similarly, if y_i is not a multiple of 8, the encoding process shall extend the number of lines to complete the bottom-most block-row. If the component is to be interleaved, the encoding process shall also extend the number of lines by one or more additional block-rows, if necessary, so that the number of block-rows is an integer multiple of V_i.

NOTE – It is recommended that any incomplete MCUs be completed by replication of the right-most column and the bottom line of each component.

For lossless processes the data unit is a sample. If the component is to be interleaved, the encoding process shall extend the number of samples, if necessary, so that the number is a multiple of H_i. Similarly, the encoding process shall extend the number of lines, if necessary, so that the number of lines is a multiple of V_i.

Any sample added by an encoding process to complete partial MCUs shall be removed by the decoding process.

A.3 DCT compression

A.3.1 Level shift

Before a non-differential frame encoding process computes the FDCT for a block of source image samples, the samples shall be level shifted to a signed representation by subtracting 2^{P-1}, where P is the precision parameter specified in B.2.2. Thus, when P = 8, the level shift is by 128; when P = 12, the level shift is by 2048.

After a non-differential frame decoding process computes the IDCT and produces a block of reconstructed image samples, an inverse level shift shall restore the samples to the unsigned representation by adding 2^{P-1} and clamping the results to the range 0 to 2^{P-1}.

A.3.2 Orientation of samples for FDCT computation

Figure A.4 shows an image component which has been partitioned into 8×8 blocks for the FDCT computations. Figure A.4 also defines the orientation of the samples within a block by showing the indices used in the FDCT equation of A.3.3.

The definitions of block partitioning and sample orientation also apply to any DCT decoding process and the output reconstructed image. Any sample added by an encoding process to complete partial MCUs shall be removed by the decoding process.

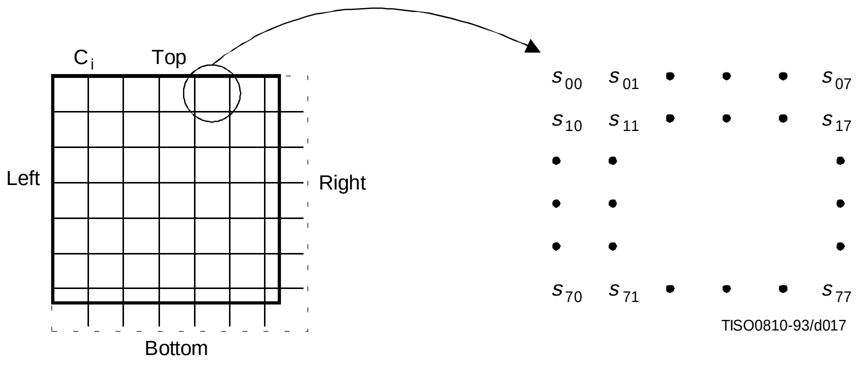

C_i
Top
Left
Right
Bottom
s_{00} s_{01} s_{07}
s_{10} s_{11} s_{17}
s_{70} s_{71} s_{77}
TISO0810-93/d017

Figure A.4 – Partition and orientation of 8 x 8 sample blocks

A.3.3 FDCT and IDCT (informative)

The following equations specify the ideal functional definition of the FDCT and the IDCT.

NOTE – These equations contain terms which cannot be represented with perfect accuracy by any real implementation. The accuracy requirements for the combined FDCT and quantization procedures are specified in Part 2 of this Specification. The accuracy requirements for the combined dequantization and IDCT procedures are also specified in Part 2 of this Specification.

$$\text{FDCT:} \qquad S_{vu} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{yx} \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16}$$

$$\text{IDCT:} \qquad s_{yx} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C_u C_v S_{vu} \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16}$$

where

$$C_u, C_v = 1/\sqrt{2} \text{ for } u, v = 0$$

$$C_u, C_v = 1 \text{ otherwise}$$

otherwise.

A.3.4 DCT coefficient quantization (informative) and dequantization (normative)

After the FDCT is computed for a block, each of the 64 resulting DCT coefficients is quantized by a uniform quantizer. The quantizer step size for each coefficient S_{vu} is the value of the corresponding element Q_{vu} from the quantization table specified by the frame parameter Tq_i (see B.2.2).

The uniform quantizer is defined by the following equation. Rounding is to the nearest integer:

$$Sq_{vu} = round\left(\frac{S_{vu}}{Q_{vu}}\right)$$

Sq_{vu} is the quantized DCT coefficient, normalized by the quantizer step size.

NOTE – This equation contains a term which may not be represented with perfect accuracy by any real implementation. The accuracy requirements for the combined FDCT and quantization procedures are specified in Part 2 of this Specification.

At the decoder, this normalization is removed by the following equation, which defines dequantization:

$$R_{vu} = Sq_{vu} \times Q_{vu}$$

NOTE – Depending on the rounding used in quantization, it is possible that the dequantized coefficient may be outside the expected range.

The relationship among samples, DCT coefficients, and quantization is illustrated in Figure A.5.

A.3.5 Differential DC encoding

After quantization, and in preparation for entropy encoding, the quantized DC coefficient Sq_{00} is treated separately from the 63 quantized AC coefficients. The value that shall be encoded is the difference (DIFF) between the quantized DC coefficient of the current block (DC_i which is also designated as Sq_{00}) and that of the previous block of the same component (PRED):

$$DIFF = DC_i - PRED$$

A.3.6 Zig-zag sequence

After quantization, and in preparation for entropy encoding, the quantized AC coefficients are converted to the zig-zag sequence. The quantized DC coefficient (coefficient zero in the array) is treated separately, as defined in A.3.5. The zig-zag sequence is specified in Figure A.6.

A.4 Point transform

For various procedures data may be optionally divided by a power of 2 by a point transform prior to coding. There are three processes which require a point transform: lossless coding, lossless differential frame coding in the hierarchical mode, and successive approximation coding in the progressive DCT mode.

In the lossless mode of operation the point transform is applied to the input samples. In the difference coding of the hierarchical mode of operation the point transform is applied to the difference between the input component samples and the reference component samples. In both cases the point transform is an integer divide by 2^{Pt}, where Pt is the value of the point transform parameter (see B.2.3).

In successive approximation coding the point transform for the AC coefficients is an integer divide by 2^{Al}, where Al is the successive approximation bit position, low (see B.2.3). The point transform for the DC coefficients is an arithmetic-shift-right by Al bits. This is equivalent to dividing by 2^{Pt} before the level shift (see A.3.1).

The output of the decoder is rescaled by multiplying by 2^{Pt}. An example of the point transform is given in K.10.

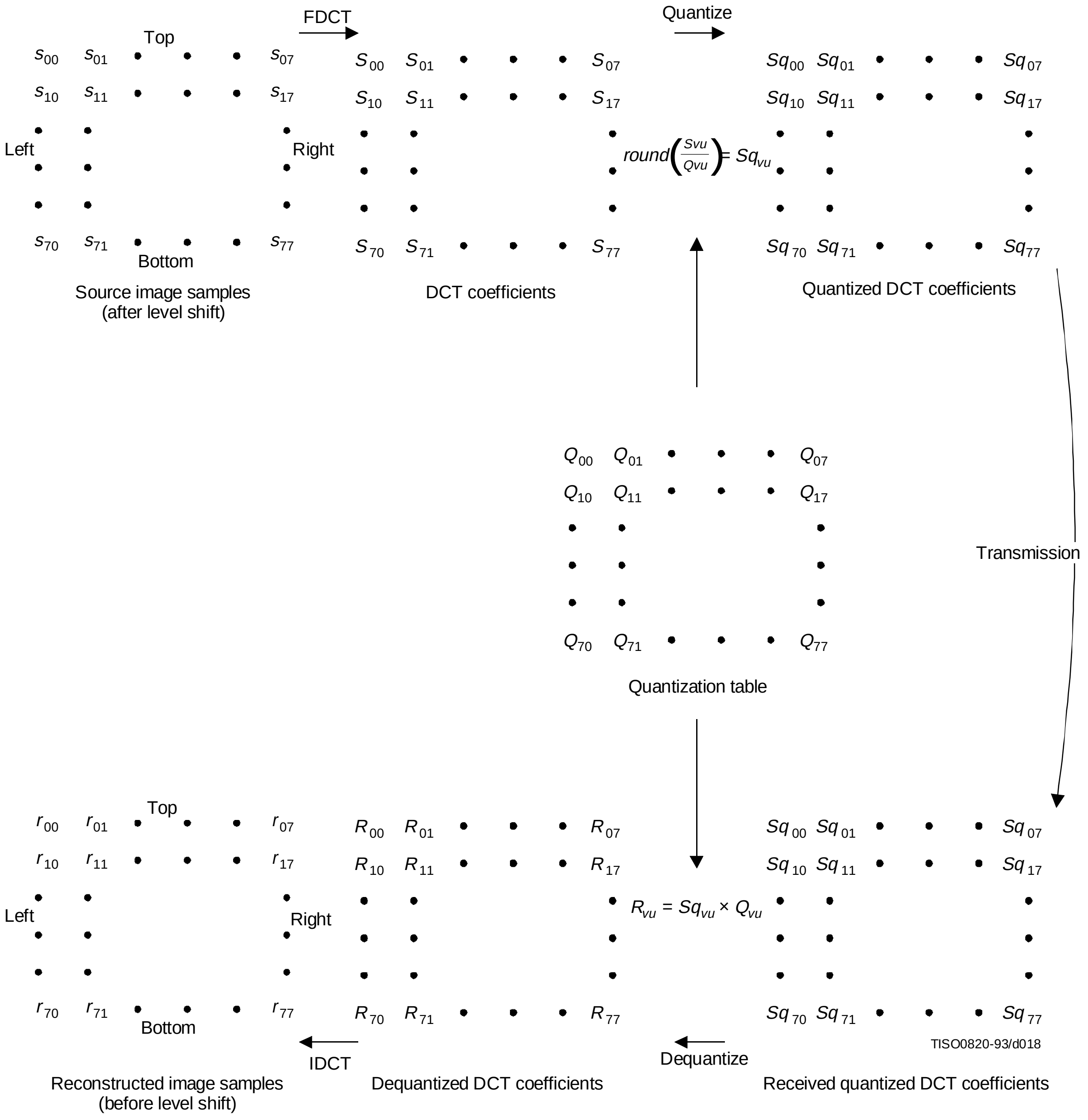

FDCT
Quantize
Top
Left
Right
Bottom
Source image samples (after level shift)
DCT coefficients
Quantized DCT coefficients
$round\left(\frac{Svu}{Qvu}\right) = Sq_{vu}$
Quantization table
Transmission
$R_{vu} = Sq_{vu} \times Q_{vu}$
Reconstructed image samples (before level shift)
Dequantized DCT coefficients
Received quantized DCT coefficients
IDCT
Dequantize
TISO0820-93/d018

Figure A.5 – Relationship between 8 × 8-block samples and DCT coefficients

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

Figure A.6 – Zig-zag sequence of quantized DCT coefficients

A.5 Arithmetic procedures in lossless and hierarchical modes of operation

In the lossless mode of operation predictions are calculated with full precision and without clamping of either overflow or underflow beyond the range of values allowed by the precision of the input. However, the division by two which is part of some of the prediction calculations shall be approximated by an arithmetic-shift-right by one bit.

The two’s complement differences which are coded in either the lossless mode of operation or the differential frame coding in the hierarchical mode of operation are calculated modulo 65 536, thereby restricting the precision of these differences to a maximum of 16 bits. The modulo values are calculated by performing the logical AND operation of the two’s complement difference with X’FFFF’. For purposes of coding, the result is still interpreted as a 16 bit two’s complement difference. Modulo 65 536 arithmetic is also used in the decoder in calculating the output from the sum of the prediction and this two’s complement difference.

Annex B

Compressed data formats

(This annex forms an integral part of this Recommendation | International Standard)

This annex specifies three compressed data formats:

a) the interchange format, specified in B.2 and B.3;
b) the abbreviated format for compressed image data, specified in B.4;
c) the abbreviated format for table-specification data, specified in B.5.

B.1 describes the constituent parts of these formats. B.1.3 and B.1.4 give the conventions for symbols and figures used in the format specifications.

B.1 General aspects of the compressed data format specifications

Structurally, the compressed data formats consist of an ordered collection of parameters, markers, and entropy-coded data segments. Parameters and markers in turn are often organized into marker segments. Because all of these constituent parts are represented with byte-aligned codes, each compressed data format consists of an ordered sequence of 8-bit bytes. For each byte, a most significant bit (MSB) and a least significant bit (LSB) are defined.

B.1.1 Constituent parts

This subclause gives a general description of each of the constituent parts of the compressed data format.

B.1.1.1 Parameters

Parameters are integers, with values specific to the encoding process, source image characteristics, and other features selectable by the application. Parameters are assigned either 4-bit, 1-byte, or 2-byte codes. Except for certain optional groups of parameters, parameters encode critical information without which the decoding process cannot properly reconstruct the image.

The code assignment for a parameter shall be an unsigned integer of the specified length in bits with the particular value of the parameter.

For parameters which are 2 bytes (16 bits) in length, the most significant byte shall come first in the compressed data's ordered sequence of bytes. Parameters which are 4 bits in length always come in pairs, and the pair shall always be encoded in a single byte. The first 4-bit parameter of the pair shall occupy the most significant 4 bits of the byte. Within any 16-, 8-, or 4-bit parameter, the MSB shall come first and LSB shall come last.

B.1.1.2 Markers

Markers serve to identify the various structural parts of the compressed data formats. Most markers start marker segments containing a related group of parameters; some markers stand alone. All markers are assigned two-byte codes: an X'FF' byte followed by a byte which is not equal to 0 or X'FF' (see Table B.1). Any marker may optionally be preceded by any number of fill bytes, which are bytes assigned code X'FF'.

NOTE – Because of this special code-assignment structure, markers make it possible for a decoder to parse the compressed data and locate its various parts without having to decode other segments of image data.

B.1.1.3 Marker assignments

All markers shall be assigned two-byte codes: a X'FF' byte followed by a second byte which is not equal to 0 or X'FF'. The second byte is specified in Table B.1 for each defined marker. An asterisk (*) indicates a marker which stands alone, that is, which is not the start of a marker segment.

Table B.1 – Marker code assignments

Code Assignment	Symbol	Description
Start Of Frame markers, non-differential, Huffman coding		
X’FFC0’	SOF_0	Baseline DCT
X’FFC1’	SOF_1	Extended sequential DCT
X’FFC2’	SOF_2	Progressive DCT
X’FFC3’	SOF_3	Lossless (sequential)
Start Of Frame markers, differential, Huffman coding		
X’FFC5’	SOF_5	Differential sequential DCT
X’FFC6’	SOF_6	Differential progressive DCT
X’FFC7’	SOF_7	Differential lossless (sequential)
Start Of Frame markers, non-differential, arithmetic coding		
X’FFC8’	JPG	Reserved for JPEG extensions
X’FFC9’	SOF_9	Extended sequential DCT
X’FFCA’	SOF_{10}	Progressive DCT
X’FFCB’	SOF_{11}	Lossless (sequential)
Start Of Frame markers, differential, arithmetic coding		
X’FFCD’	SOF_{13}	Differential sequential DCT
X’FFCE’	SOF_{14}	Differential progressive DCT
X’FFCF’	SOF_{15}	Differential lossless (sequential)
Huffman table specification		
X’FFC4’	DHT	Define Huffman table(s)
Arithmetic coding conditioning specification		
X’FFCC’	DAC	Define arithmetic coding conditioning(s)
Restart interval termination		
X’FFD0’ through X’FFD7’	RST_m*	Restart with modulo 8 count “m”
Other markers		
X’FFD8’	SOI*	Start of image
X’FFD9’	EOI*	End of image
X’FFDA’	SOS	Start of scan
X’FFDB’	DQT	Define quantization table(s)
X’FFDC’	DNL	Define number of lines
X’FFDD’	DRI	Define restart interval
X’FFDE’	DHP	Define hierarchical progression
X’FFDF’	EXP	Expand reference component(s)
X’FFE0’ through X’FFEF’	APP_n	Reserved for application segments
X’FFF0’ through X’FFFD’	JPG_n	Reserved for JPEG extensions
X’FFFE’	COM	Comment
Reserved markers		
X’FF01’	TEM*	For temporary private use in arithmetic coding
X’FF02’ through X’FFBF’	RES	Reserved

B.1.1.4 Marker segments

A marker segment consists of a marker followed by a sequence of related parameters. The first parameter in a marker segment is the two-byte length parameter. This length parameter encodes the number of bytes in the marker segment, including the length parameter and excluding the two-byte marker. The marker segments identified by the SOF and SOS marker codes are referred to as headers: the frame header and the scan header respectively.

B.1.1.5 Entropy-coded data segments

An entropy-coded data segment contains the output of an entropy-coding procedure. It consists of an integer number of bytes, whether the entropy-coding procedure used is Huffman or arithmetic.

NOTES

1 Making entropy-coded segments an integer number of bytes is performed as follows: for Huffman coding, 1-bits are used, if necessary, to pad the end of the compressed data to complete the final byte of a segment. For arithmetic coding, byte alignment is performed in the procedure which terminates the entropy-coded segment (see D.1.8).

2 In order to ensure that a marker does not occur within an entropy-coded segment, any X'FF' byte generated by either a Huffman or arithmetic encoder, or an X'FF' byte that was generated by the padding of 1-bits described in NOTE 1 above, is followed by a "stuffed" zero byte (see D.1.6 and F.1.2.3).

B.1.2 Syntax

In B.2 and B.3 the interchange format syntax is specified. For the purposes of this Specification, the syntax specification consists of:

- the required ordering of markers, parameters, and entropy-coded segments;
- identification of optional or conditional constituent parts;
- the name, symbol, and definition of each marker and parameter;
- the allowed values of each parameter;
- any restrictions on the above which are specific to the various coding processes.

The ordering of constituent parts and the identification of which are optional or conditional is specified by the syntax figures in B.2 and B.3. Names, symbols, definitions, allowed values, conditions, and restrictions are specified immediately below each syntax figure.

B.1.3 Conventions for syntax figures

The syntax figures in B.2 and B.3 are a part of the interchange format specification. The following conventions, illustrated in Figure B.1, apply to these figures:

- **parameter/marker indicator:** A thin-lined box encloses either a marker or a single parameter;
- **segment indicator:** A thick-lined box encloses either a marker segment, an entropy-coded data segment, or combinations of these;
- **parameter length indicator:** The width of a thin-lined box is proportional to the parameter length (4, 8, or 16 bits, shown as E, B, and D respectively in Figure B.1) of the marker or parameter it encloses; the width of thick-lined boxes is not meaningful;
- **optional/conditional indicator:** Square brackets indicate that a marker or marker segment is only optionally or conditionally present in the compressed image data;
- **ordering:** In the interchange format a parameter or marker shown in a figure precedes all of those shown to its right, and follows all of those shown to its left;
- **entropy-coded data indicator:** Angled brackets indicate that the entity enclosed has been entropy encoded.

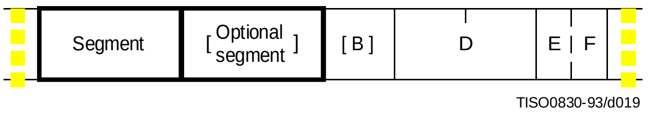


Figure B.1 – Syntax notation conventions

B.1.4 Conventions for symbols, code lengths, and values

Following each syntax figure in B.2 and B.3, the symbol, name, and definition for each marker and parameter shown in the figure are specified. For each parameter, the length and allowed values are also specified in tabular form.

The following conventions apply to symbols for markers and parameters:

- all marker symbols have three upper-case letters, and some also have a subscript. Examples: SOI, SOF_n;
- all parameter symbols have one upper-case letter; some also have one lower-case letter and some have subscripts. Examples: Y, Nf, H_i, Tq_i.

B.2 General sequential and progressive syntax

This clause specifies the interchange format syntax which applies to all coding processes for sequential DCT-based, progressive DCT-based, and lossless modes of operation.

B.2.1 High-level syntax

Figure B.2 specifies the order of the high-level constituent parts of the interchange format for all non-hierarchical encoding processes specified in this Specification.

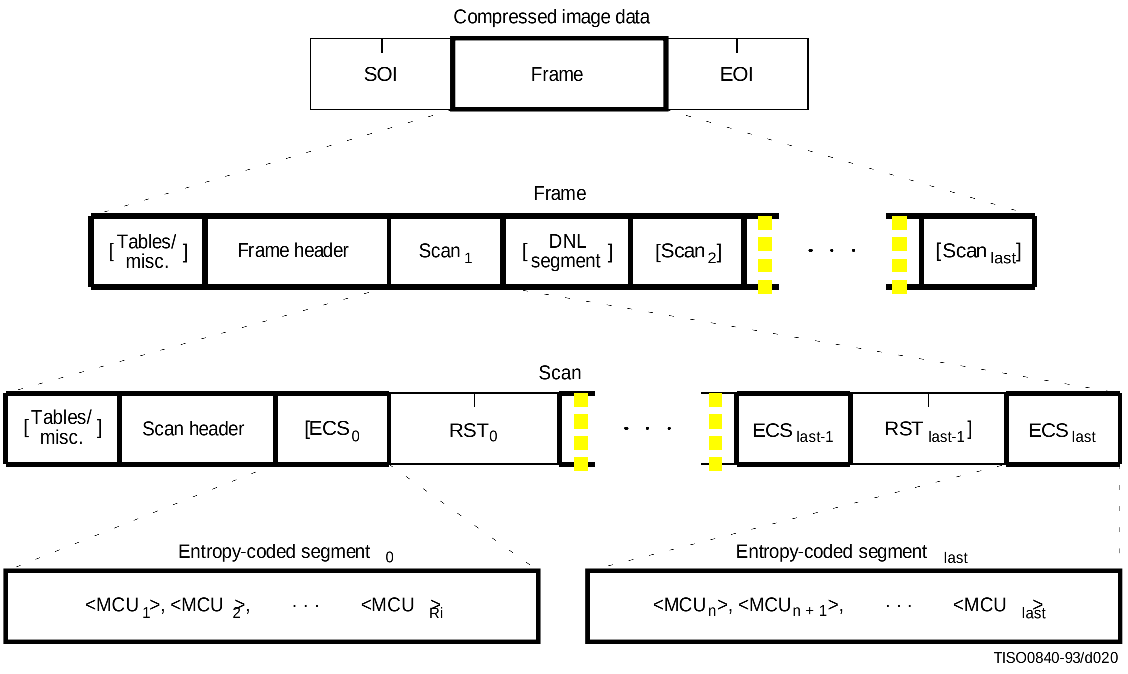


Figure B.2 – Syntax for sequential DCT-based, progressive DCT-based, and lossless modes of operation

The three markers shown in Figure B.2 are defined as follows:

SOI: Start of image marker – Marks the start of a compressed image represented in the interchange format or abbreviated format.

EOI: End of image marker – Marks the end of a compressed image represented in the interchange format or abbreviated format.

RST_m: Restart marker – A conditional marker which is placed between entropy-coded segments only if restart is enabled. There are 8 unique restart markers (m = 0 - 7) which repeat in sequence from 0 to 7, starting with zero for each scan, to provide a modulo 8 restart interval count.

The top level of Figure B.2 specifies that the non-hierarchical interchange format shall begin with an SOI marker, shall contain one frame, and shall end with an EOI marker.

The second level of Figure B.2 specifies that a frame shall begin with a frame header and shall contain one or more scans. A frame header may be preceded by one or more table-specification or miscellaneous marker segments as specified in B.2.4. If a DNL segment (see B.2.5) is present, it shall immediately follow the first scan.

For sequential DCT-based and lossless processes each scan shall contain from one to four image components. If two to four components are contained within a scan, they shall be interleaved within the scan. For progressive DCT-based processes each image component is only partially contained within any one scan. Only the first scan(s) for the components (which contain only DC coefficient data) may be interleaved.

The third level of Figure B.2 specifies that a scan shall begin with a scan header and shall contain one or more entropy-coded data segments. Each scan header may be preceded by one or more table-specification or miscellaneous marker segments. If restart is not enabled, there shall be only one entropy-coded segment (the one labeled "last"), and no restart markers shall be present. If restart is enabled, the number of entropy-coded segments is defined by the size of the image and the defined restart interval. In this case, a restart marker shall follow each entropy-coded segment except the last one.

The fourth level of Figure B.2 specifies that each entropy-coded segment is comprised of a sequence of entropy-coded MCUs. If restart is enabled and the restart interval is defined to be Ri, each entropy-coded segment except the last one shall contain Ri MCUs. The last one shall contain whatever number of MCUs completes the scan.

Figure B.2 specifies the locations where table-specification segments **may** be present. However, this Specification hereby specifies that the interchange format **shall** contain all table-specification data necessary for decoding the compressed image. Consequently, the required table-specification data **shall** be present at one or more of the allowed locations.

B.2.2 Frame header syntax

Figure B.3 specifies the frame header which shall be present at the start of a frame. This header specifies the source image characteristics (see A.1), the components in the frame, and the sampling factors for each component, and specifies the destinations from which the quantized tables to be used with each component are retrieved.

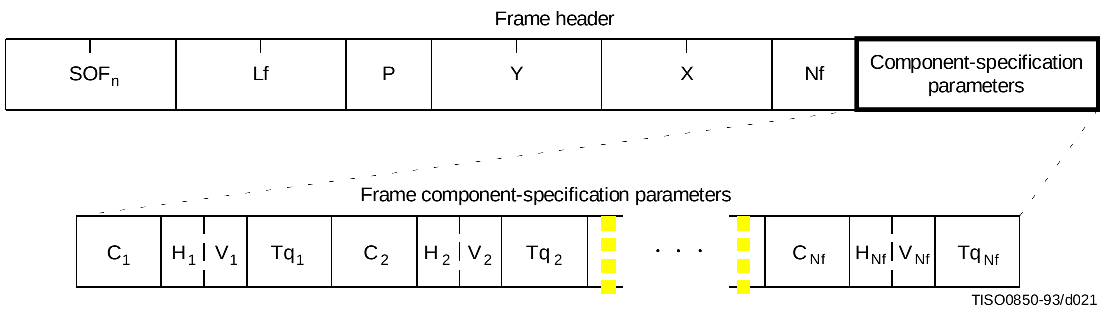


Figure B.3 – Frame header syntax

The markers and parameters shown in Figure B.3 are defined below. The size and allowed values of each parameter are given in Table B.2. In Table B.2 (and similar tables which follow), value choices are separated by commas (e.g. 8, 12) and inclusive bounds are separated by dashes (e.g. 0 - 3).

SOF$_n$: Start of frame marker – Marks the beginning of the frame parameters. The subscript n identifies whether the encoding process is baseline sequential, extended sequential, progressive, or lossless, as well as which entropy encoding procedure is used.

SOF$_0$: Baseline DCT

SOF$_1$: Extended sequential DCT, Huffman coding

SOF$_2$: Progressive DCT, Huffman coding

SOF_3: Lossless (sequential), Huffman coding

SOF_9: Extended sequential DCT, arithmetic coding

SOF_{10}: Progressive DCT, arithmetic coding

SOF_{11}: Lossless (sequential), arithmetic coding

Lf: Frame header length – Specifies the length of the frame header shown in Figure B.3 (see B.1.1.4).

P: Sample precision – Specifies the precision in bits for the samples of the components in the frame.

Y: Number of lines – Specifies the maximum number of lines in the source image. This shall be equal to the number of lines in the component with the maximum number of vertical samples (see A.1.1). Value 0 indicates that the number of lines shall be defined by the DNL marker and parameters at the end of the first scan (see B.2.5).

X: Number of samples per line – Specifies the maximum number of samples per line in the source image. This shall be equal to the number of samples per line in the component with the maximum number of horizontal samples (see A.1.1).

Nf: Number of image components in frame – Specifies the number of source image components in the frame. The value of Nf shall be equal to the number of sets of frame component specification parameters (C_i, H_i, V_i, and Tq_i) present in the frame header.

C_i: Component identifier – Assigns a unique label to the *i*th component in the sequence of frame component specification parameters. These values shall be used in the scan headers to identify the components in the scan. The value of C_i shall be different from the values of C_1 through C_{i-1}.

H_i: Horizontal sampling factor – Specifies the relationship between the component horizontal dimension and maximum image dimension X (see A.1.1); also specifies the number of horizontal data units of component C_i in each MCU, when more than one component is encoded in a scan.

V_i: Vertical sampling factor – Specifies the relationship between the component vertical dimension and maximum image dimension Y (see A.1.1); also specifies the number of vertical data units of component C_i in each MCU, when more than one component is encoded in a scan.

Tq_i: Quantization table destination selector – Specifies one of four possible quantization table destinations from which the quantization table to use for dequantization of DCT coefficients of component C_i is retrieved. If the decoding process uses the dequantization procedure, this table shall have been installed in this destination by the time the decoder is ready to decode the scan(s) containing component C_i. The destination shall not be re-specified, or its contents changed, until all scans containing C_i have been completed.

Table B.2 – Frame header parameter sizes and values

Parameter	Size (bits)	Values			
		Sequential DCT		Progressive DCT	Lossless
		Baseline	Extended		
Lf	16	$8 + 3 \times Nf$			
P	8	8	8, 12	8, 12	2-16
Y	16	0-65 535			
X	16	1-65 535			
Nf	8	1-255	1-255	1-4	1-255
C_i	8	0-255			
H_i	4	1-4			
V_i	4	1-4			
Tq_i	8	0-3	0-3	0-3	0

B.2.3 Scan header syntax

Figure B.4 specifies the scan header which shall be present at the start of a scan. This header specifies which component(s) are contained in the scan, specifies the destinations from which the entropy tables to be used with each component are retrieved, and (for the progressive DCT) which part of the DCT quantized coefficient data is contained in the scan. For lossless processes the scan parameters specify the predictor and the point transform.

NOTE – If there is only one image component present in a scan, that component is, by definition, non-interleaved. If there is more than one image component present in a scan, the components present are, by definition, interleaved.

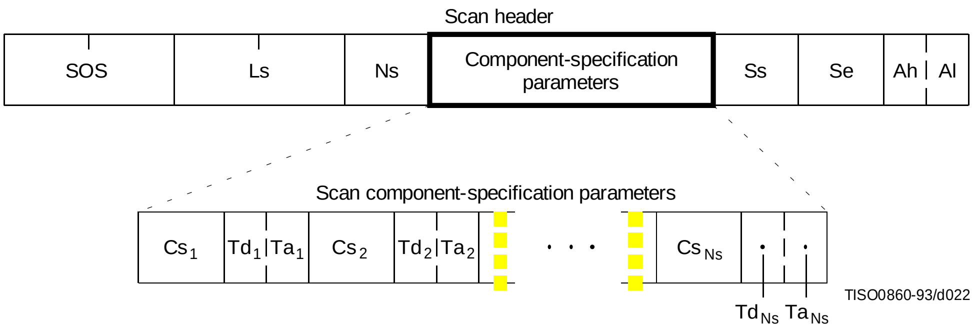

Scan header
SOS
Ls
Ns
Component-specification parameters
Ss
Se
Ah
Al
Scan component-specification parameters
Cs1
Td1
Ta1
Cs2
Td2
Ta2
Cs Ns
TdNs
TaNs
TISO0860-93/d022

Figure B.4 – Scan header syntax

The marker and parameters shown in Figure B.4 are defined below. The size and allowed values of each parameter are given in Table B.3.

SOS: Start of scan marker – Marks the beginning of the scan parameters.

Ls: Scan header length – Specifies the length of the scan header shown in Figure B.4 (see B.1.1.4).

Ns: Number of image components in scan – Specifies the number of source image components in the scan. The value of Ns shall be equal to the number of sets of scan component specification parameters (Cs_j, Td_j, and Ta_j) present in the scan header.

Cs_j: Scan component selector – Selects which of the Nf image components specified in the frame parameters shall be the *j*th component in the scan. Each Cs_j shall match one of the C_i values specified in the frame header, and the ordering in the scan header shall follow the ordering in the frame header. If Ns > 1, the order of interleaved components in the MCU is Cs_1 first, Cs_2 second, etc. If Ns > 1, the following restriction shall be placed on the image components contained in the scan:

$$\sum_{j=1}^{N_s} H_j \times V_j \leq 10,$$

where H_j and V_j are the horizontal and vertical sampling factors for scan component j. These sampling factors are specified in the frame header for component i, where i is the frame component specification index for which frame component identifier C_i matches scan component selector Cs_j.

As an example, consider an image having 3 components with maximum dimensions of 512 lines and 512 samples per line, and with the following sampling factors:

Component 0	$H_0 = 4,$	$V_0 = 1$
Component 1	$H_1 = 1,$	$V_1 = 2$
Component 2	$H_2 = 2$	$V_2 = 2$

Then the summation of $H_j \times V_j$ is $(4 \times 1) + (1 \times 2) + (2 \times 2) = 10$.

The value of Cs_j shall be different from the values of Cs_1 to Cs_{j-1}.

Td_j: DC entropy coding table destination selector – Specifies one of four possible DC entropy coding table destinations from which the entropy table needed for decoding of the DC coefficients of component Cs_j is retrieved. The DC entropy table shall have been installed in this destination (see B.2.4.2 and B.2.4.3) by the time the decoder is ready to decode the current scan. This parameter specifies the entropy coding table destination for the lossless processes.

Ta_j: AC entropy coding table destination selector – Specifies one of four possible AC entropy coding table destinations from which the entropy table needed for decoding of the AC coefficients of component Cs_j is retrieved. The AC entropy table selected shall have been installed in this destination (see B.2.4.2 and B.2.4.3) by the time the decoder is ready to decode the current scan. This parameter is zero for the lossless processes.

Ss: Start of spectral or predictor selection – In the DCT modes of operation, this parameter specifies the first DCT coefficient in each block in zig-zag order which shall be coded in the scan. This parameter shall be set to zero for the sequential DCT processes. In the lossless mode of operations this parameter is used to select the predictor.

Se: End of spectral selection – Specifies the last DCT coefficient in each block in zig-zag order which shall be coded in the scan. This parameter shall be set to 63 for the sequential DCT processes. In the lossless mode of operations this parameter has no meaning. It shall be set to zero.

Ah: Successive approximation bit position high – This parameter specifies the point transform used in the preceding scan (i.e. successive approximation bit position low in the preceding scan) for the band of coefficients specified by Ss and Se. This parameter shall be set to zero for the first scan of each band of coefficients. In the lossless mode of operations this parameter has no meaning. It shall be set to zero.

Al: Successive approximation bit position low or point transform – In the DCT modes of operation this parameter specifies the point transform, i.e. bit position low, used before coding the band of coefficients specified by Ss and Se. This parameter shall be set to zero for the sequential DCT processes. In the lossless mode of operations, this parameter specifies the point transform, Pt.

The entropy coding table destination selectors, Td_j and Ta_j, specify either Huffman tables (in frames using Huffman coding) or arithmetic coding tables (in frames using arithmetic coding). In the latter case the entropy coding table destination selector specifies both an arithmetic coding conditioning table destination and an associated statistics area.

Table B.3 – Scan header parameter size and values

Parameter	Size (bits)	Values			
		Sequential DCT		Progressive DCT	Lossless
		Baseline	Extended		
Ls	16	$6 + 2 \times Ns$			
Ns	8	1-4			
Cs_j	8	0-255[a)]			
Td_j	4	0-1	0-3	0-3	0-3
Ta_j	4	0-1	0-3	0-3	0
Ss	8	0	0	0-63	1-7[b)]
Se	8	63	63	Ss-63[c)]	0
Ah	4	0	0	0-13	0
Al	4	0	0	0-13	0-15

a) Cs_j shall be a member of the set of C_i specified in the frame header.

b) 0 for lossless differential frames in the hierarchical mode (see B.3).

c) 0 if Ss equals zero.

B.2.4 Table-specification and miscellaneous marker segment syntax

Figure B.5 specifies that, at the places indicated in Figure B.2, any of the table-specification segments or miscellaneous marker segments specified in B.2.4.1 through B.2.4.6 may be present in any order and with no limit on the number of segments.

If any table specification for a particular destination occurs in the compressed image data, it shall replace any previous table specified for this destination, and shall be used whenever this destination is specified in the remaining scans in the frame or subsequent images represented in the abbreviated format for compressed image data. If a table specification for a given destination occurs more than once in the compressed image data, each specification shall replace the previous specification. The quantization table specification shall not be altered between progressive DCT scans of a given component.

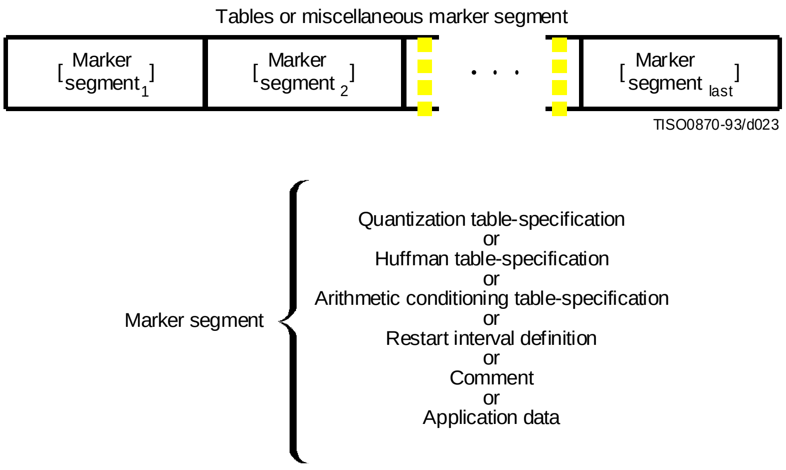

Tables or miscellaneous marker segment
[Marker segment 1]
[Marker segment 2]
. . .
[Marker segment last]
TISO0870-93/d023
Marker segment
Quantization table-specification
or
Huffman table-specification
or
Arithmetic conditioning table-specification
or
Restart interval definition
or
Comment
or
Application data

Figure B.5 – Tables/miscellaneous marker segment syntax

B.2.4.1 Quantization table-specification syntax

Figure B.6 specifies the marker segment which defines one or more quantization tables.

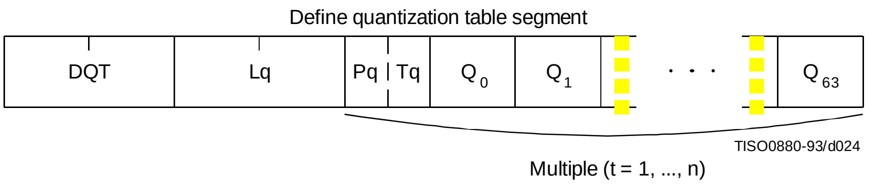


Figure B.6 – Quantization table syntax

The marker and parameters shown in Figure B.6 are defined below. The size and allowed values of each parameter are given in Table B.4.

DQT: Define quantization table marker – Marks the beginning of quantization table-specification parameters.

Lq: Quantization table definition length – Specifies the length of all quantization table parameters shown in Figure B.6 (see B.1.1.4).

Pq: Quantization table element precision – Specifies the precision of the Q_k values. Value 0 indicates 8-bit Q_k values; value 1 indicates 16-bit Q_k values. Pq shall be zero for 8 bit sample precision P (see B.2.2).

Tq: Quantization table destination identifier – Specifies one of four possible destinations at the decoder into which the quantization table shall be installed.

Q_k: Quantization table element – Specifies the *k*th element out of 64 elements, where *k* is the index in the zig-zag ordering of the DCT coefficients. The quantization elements shall be specified in zig-zag scan order.

Table B.4 – Quantization table-specification parameter sizes and values

Parameter	Size (bits)	Values			
		Sequential DCT		Progressive DCT	Lossless
		Baseline	Extended		
Lq	16	$2 + \sum_{t=1}^{n} (65 + 64 \times Pq(t))$			Undefined
Pq	4	0	0, 1	0, 1	Undefined
Tq	4	0-3			Undefined
Q_k	8, 16	1-255, 1-65 535			Undefined

The value n in Table B.4 is the number of quantization tables specified in the DQT marker segment.

Once a quantization table has been defined for a particular destination, it replaces the previous tables stored in that destination and shall be used, when referenced, in the remaining scans of the current image and in subsequent images represented in the abbreviated format for compressed image data. If a table has never been defined for a particular destination, then when this destination is specified in a frame header, the results are unpredictable.

An 8-bit DCT-based process shall not use a 16-bit precision quantization table.

B.2.4.2 Huffman table-specification syntax

Figure B.7 specifies the marker segment which defines one or more Huffman table specifications.

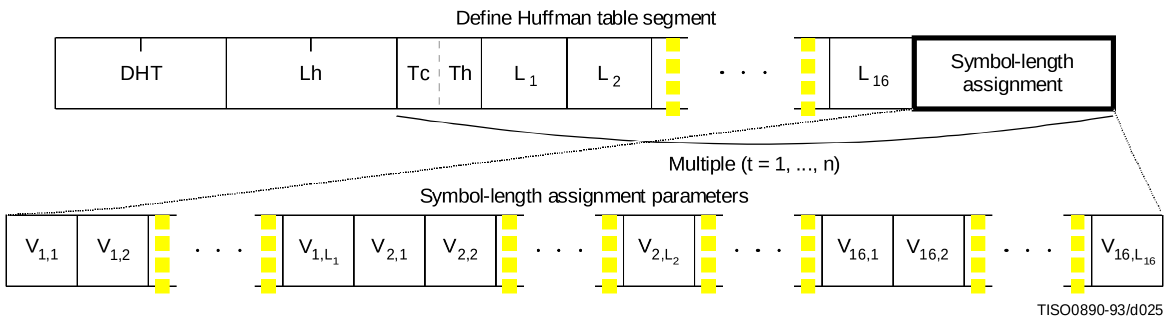


Figure B.7 – Huffman table syntax

The marker and parameters shown in Figure B.7 are defined below. The size and allowed values of each parameter are given in Table B.5.

DHT: Define Huffman table marker – Marks the beginning of Huffman table definition parameters.

Lh: Huffman table definition length – Specifies the length of all Huffman table parameters shown in Figure B.7 (see B.1.1.4).

Tc: Table class – 0 = DC table or lossless table, 1 = AC table.

Th: Huffman table destination identifier – Specifies one of four possible destinations at the decoder into which the Huffman table shall be installed.

L_i: Number of Huffman codes of length i – Specifies the number of Huffman codes for each of the 16 possible lengths allowed by this Specification. L_i's are the elements of the list BITS.

$V_{i,j}$: Value associated with each Huffman code – Specifies, for each i, the value associated with each Huffman code of length i. The meaning of each value is determined by the Huffman coding model. The $V_{i,j}$'s are the elements of the list HUFFVAL.

Table B.5 – Huffman table specification parameter sizes and values

Parameter	Size (bits)	Values			
		Sequential DCT		Progressive DCT	Lossless
		Baseline	Extended		
Lh	16	$2 + \sum_{t=1}^{n} (17 + m_t)$			
Tc	4	0, 1			0
Th	4	0, 1	0-3		
L_i	8	0-255			
$V_{i,j}$	8	0-255			

The value n in Table B.5 is the number of Huffman tables specified in the DHT marker segment. The value m_t is the number of parameters which follow the 16 $L_i(t)$ parameters for Huffman table t, and is given by:

$$m_t = \sum_{i=1}^{16} L_i$$

In general, m_t is different for each table.

Once a Huffman table has been defined for a particular destination, it replaces the previous tables stored in that destination and shall be used when referenced, in the remaining scans of the current image and in subsequent images represented in the abbreviated format for compressed image data. If a table has never been defined for a particular destination, then when this destination is specified in a scan header, the results are unpredictable.

B.2.4.3 Arithmetic conditioning table-specification syntax

Figure B.8 specifies the marker segment which defines one or more arithmetic coding conditioning table specifications. These replace the default arithmetic coding conditioning tables established by the SOI marker for arithmetic coding processes. (See F.1.4.4.1.4 and F.1.4.4.2.1.)

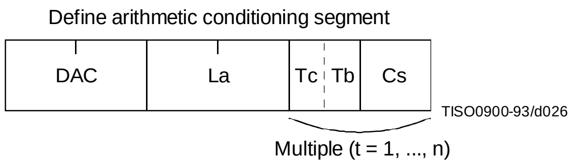


Figure B.8 – Arithmetic conditioning table-specification syntax

The marker and parameters shown in Figure B.8 are defined below. The size and allowed values of each parameter are given in Table B.6.

DAC: Define arithmetic coding conditioning marker – Marks the beginning of the definition of arithmetic coding conditioning parameters.

La: Arithmetic coding conditioning definition length – Specifies the length of all arithmetic coding conditioning parameters shown in Figure B.8 (see B.1.1.4).

Tc: Table class – 0 = DC table or lossless table, 1 = AC table.

Tb: Arithmetic coding conditioning table destination identifier – Specifies one of four possible destinations at the decoder into which the arithmetic coding conditioning table shall be installed.

Cs: Conditioning table value – Value in either the AC or the DC (and lossless) conditioning table. A single value of Cs shall follow each value of Tb. For AC conditioning tables Tc shall be one and Cs shall contain a value of Kx in the range $1 \leq Kx \leq 63$. For DC (and lossless) conditioning tables Tc shall be zero and Cs shall contain two 4-bit parameters, U and L. U and L shall be in the range $0 \leq L \leq U \leq 15$ and the value of Cs shall be $L + 16 \times U$.

The value n in Table B.6 is the number of arithmetic coding conditioning tables specified in the DAC marker segment. The parameters L and U are the lower and upper conditioning bounds used in the arithmetic coding procedures defined for DC coefficient coding and lossless coding. The separate value range 1-63 listed for DCT coding is the Kx conditioning used in AC coefficient coding.

Table B.6 – Arithmetic coding conditioning table-specification parameter sizes and values

Parameter	Size (bits)	Values			
		Sequential DCT		Progressive DCT	Lossless
		Baseline	Extended		
La	16	Undefined	2 + 2 × n		
Tc	4	Undefined	0, 1		0
Tb	4	Undefined	0-3		
Cs	8	Undefined	0-255 (Tc = 0), 1-63 (Tc = 1)		0-255

B.2.4.4 Restart interval definition syntax

Figure B.9 specifies the marker segment which defines the restart interval.

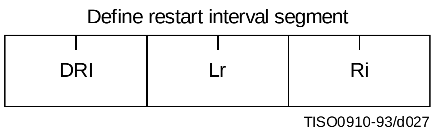


Figure B.9 – Restart interval definition syntax

The marker and parameters shown in Figure B.9 are defined below. The size and allowed values of each parameter are given in Table B.7.

DRI: Define restart interval marker – Marks the beginning of the parameters which define the restart interval.

Lr: Define restart interval segment length – Specifies the length of the parameters in the DRI segment shown in Figure B.9 (see B.1.1.4).

Ri: Restart interval – Specifies the number of MCU in the restart interval.

In Table B.7 the value n is the number of rows of MCU in the restart interval. The value MCUR is the number of MCU required to make up one line of samples of each component in the scan. The SOI marker disables the restart intervals. A DRI marker segment with Ri nonzero shall be present to enable restart interval processing for the following scans. A DRI marker segment with Ri equal to zero shall disable restart intervals for the following scans.

Table B.7 – Define restart interval segment parameter sizes and values

Parameter	Size (bits)	Values			
		Sequential DCT		Progressive DCT	Lossless
		Baseline	Extended		
Lr	16	4			
Ri	16	0-65 535			$n \times$ MCUR

B.2.4.5 Comment syntax

Figure B.10 specifies the marker segment structure for a comment segment.

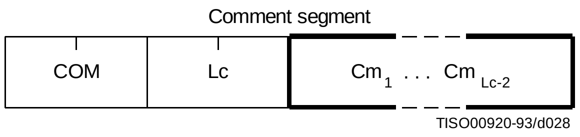


Figure B.10 – Comment segment syntax

The marker and parameters shown in Figure B.10 are defined below. The size and allowed values of each parameter are given in Table B.8.

COM: Comment marker – Marks the beginning of a comment.

Lc: Comment segment length – Specifies the length of the comment segment shown in Figure B.10 (see B.1.1.4).

Cm_i: Comment byte – The interpretation is left to the application.

Table B.8 – Comment segment parameter sizes and values

Parameter	Size (bits)	Values			
		Sequential DCT		Progressive DCT	Lossless
		Baseline	Extended		
Lc	16	2-65 535			
Cm_i	8	0-255			

B.2.4.6 Application data syntax

Figure B.11 specifies the marker segment structure for an application data segment.

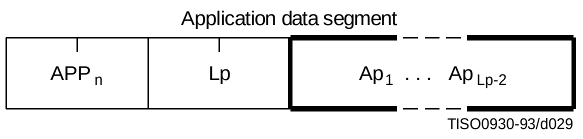


Figure B.11 – Application data syntax

The marker and parameters shown in Figure B.11 are defined below. The size and allowed values of each parameter are given in Table B.9.

APP_n: Application data marker – Marks the beginning of an application data segment.

Lp: Application data segment length – Specifies the length of the application data segment shown in Figure B.11 (see B.1.1.4).

Ap_i: Application data byte – The interpretation is left to the application.

The APP_n (Application) segments are reserved for application use. Since these segments may be defined differently for different applications, they should be removed when the data are exchanged between application environments.

Table B.9 – Application data segment parameter sizes and values

Parameter	Size (bits)	Values			
		Sequential DCT		Progressive DCT	Lossless
		Baseline	Extended		
Lp	16	2-65 535			
Ap_i	8	0-255			

B.2.5 Define number of lines syntax

Figure B.12 specifies the marker segment for defining the number of lines. The DNL (Define Number of Lines) segment provides a mechanism for defining or redefining the number of lines in the frame (the Y parameter in the frame header) at the end of the first scan. The value specified shall be consistent with the number of MCU-rows encoded in the first scan. This segment, if used, shall only occur at the end of the first scan, and only after coding of an integer number of MCU-rows. This marker segment is mandatory if the number of lines (Y) specified in the frame header has the value zero.

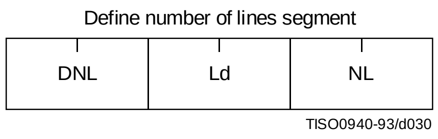


Figure B.12 – Define number of lines syntax

The marker and parameters shown in Figure B.12 are defined below. The size and allowed values of each parameter are given in Table B.10.

DNL: Define number of lines marker – Marks the beginning of the define number of lines segment.

Ld: Define number of lines segment length – Specifies the length of the define number of lines segment shown in Figure B.12 (see B.1.1.4).

NL: Number of lines – Specifies the number of lines in the frame (see definition of Y in B.2.2).

Table B.10 – Define number of lines segment parameter sizes and values

Parameter	Size (bits)	Values			
		Sequential DCT		Progressive DCT	Lossless
		Baseline	Extended		
Ld	16	4			
NL	16	1-65 535[a)]			
a) The value specified shall be consistent with the number of lines coded at the point where the DNL segment terminates the compressed data segment.					

B.3 Hierarchical syntax

B.3.1 High level hierarchical mode syntax

Figure B.13 specifies the order of the high level constituent parts of the interchange format for hierarchical encoding processes.

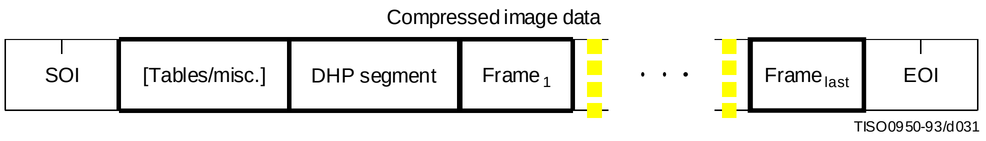


Figure B.13 – Syntax for the hierarchical mode of operation

Hierarchical mode syntax requires a DHP marker segment that appears before the non-differential frame or frames. The hierarchical mode compressed image data may include EXP marker segments and differential frames which shall follow the initial non-differential frame. The frame structure in hierarchical mode is identical to the frame structure in non-hierarchical mode.

The non-differential frames in the hierarchical sequence shall use one of the coding processes specified for SOF_n markers: SOF_0, SOF_1, SOF_2, SOF_3, SOF_9, SOF_{10} and SOF_{11}. The differential frames shall use one of the processes specified for SOF_5, SOF_6, SOF_7, SOF_{13}, SOF_{14} and SOF_{15}. The allowed combinations of SOF markers within one hierarchical sequence are specified in Annex J.

The sample precision (P) shall be constant for all frames and have the identical value as that coded in the DHP marker segment. The number of samples per line (X) for all frames shall not exceed the value coded in the DHP marker segment. If the number of lines (Y) is non-zero in the DHP marker segment, then the number of lines for all frames shall not exceed the value in the DHP marker segment.

B.3.2 DHP segment syntax

The DHP segment defines the image components, size, and sampling factors for the completed hierarchical sequence of frames. The DHP segment shall precede the first frame; a single DHP segment shall occur in the compressed image data.

The DHP segment structure is identical to the frame header syntax, except that the DHP marker is used instead of the SOF_n marker. The figures and description of B.2.2 then apply, except that the quantization table destination selector parameter shall be set to zero in the DHP segment.

B.3.3 EXP segment syntax

Figure B.14 specifies the marker segment structure for the EXP segment. The EXP segment shall be present if (and only if) expansion of the reference components is required either horizontally or vertically. The EXP segment parameters apply only to the next frame (which shall be a differential frame) in the image. If required, the EXP segment shall be one of the table-specification segments or miscellaneous marker segments preceding the frame header; the EXP segment shall not be one of the table-specification segments or miscellaneous marker segments preceding a scan header or a DHP marker segment.

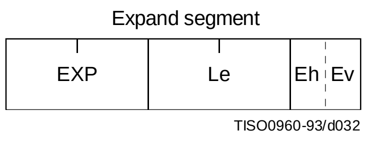


Figure B.14 – Syntax of the expand segment

The marker and parameters shown in Figure B.14 are defined below. The size and allowed values of each parameter are given in Table B.11.

EXP: Expand reference components marker – Marks the beginning of the expand reference components segment.

Le: Expand reference components segment length – Specifies the length of the expand reference components segment (see B.1.1.4).

Eh: Expand horizontally – If one, the reference components shall be expanded horizontally by a factor of two. If horizontal expansion is not required, the value shall be zero.

Ev: Expand vertically – If one, the reference components shall be expanded vertically by a factor of two. If vertical expansion is not required, the value shall be zero.

Both Eh and Ev shall be one if expansion is required both horizontally and vertically.

Table B.11 – Expand segment parameter sizes and values

Parameter	Size (bits)	Values			
		Sequential DCT		Progressive DCT	Lossless
		Baseline	Extended		
Le	16	3			
Eh	4	0, 1			
Ev	4	0, 1			

B.4 Abbreviated format for compressed image data

Figure B.2 shows the high-level constituent parts of the interchange format. This format includes all table specifications required for decoding. If an application environment provides methods for table specification other than by means of the compressed image data, some or all of the table specifications may be omitted. Compressed image data which is missing any table specification data required for decoding has the abbreviated format.

B.5 Abbreviated format for table-specification data

Figure B.2 shows the high-level constituent parts of the interchange format. If no frames are present in the compressed image data, the only purpose of the compressed image data is to convey table specifications or miscellaneous marker segments defined in B.2.4.1, B.2.4.2, B.2.4.5, and B.2.4.6. In this case the compressed image data has the abbreviated format for table specification data (see Figure B.15).

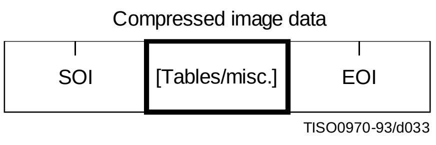


Figure B.15 – Abbreviated format for table-specification data syntax

B.6 Summary

The order of the constituent parts of interchange format and all marker segment structures is summarized in Figures B.16 and B.17. Note that in Figure B.16 double-lined boxes enclose marker segments. In Figures B.16 and B.17 thick-lined boxes enclose only markers.

The EXP segment can be mixed with the other tables/miscellaneous marker segments preceding the frame header but not with the tables/miscellaneous marker segments preceding the DHP segment or the scan header.

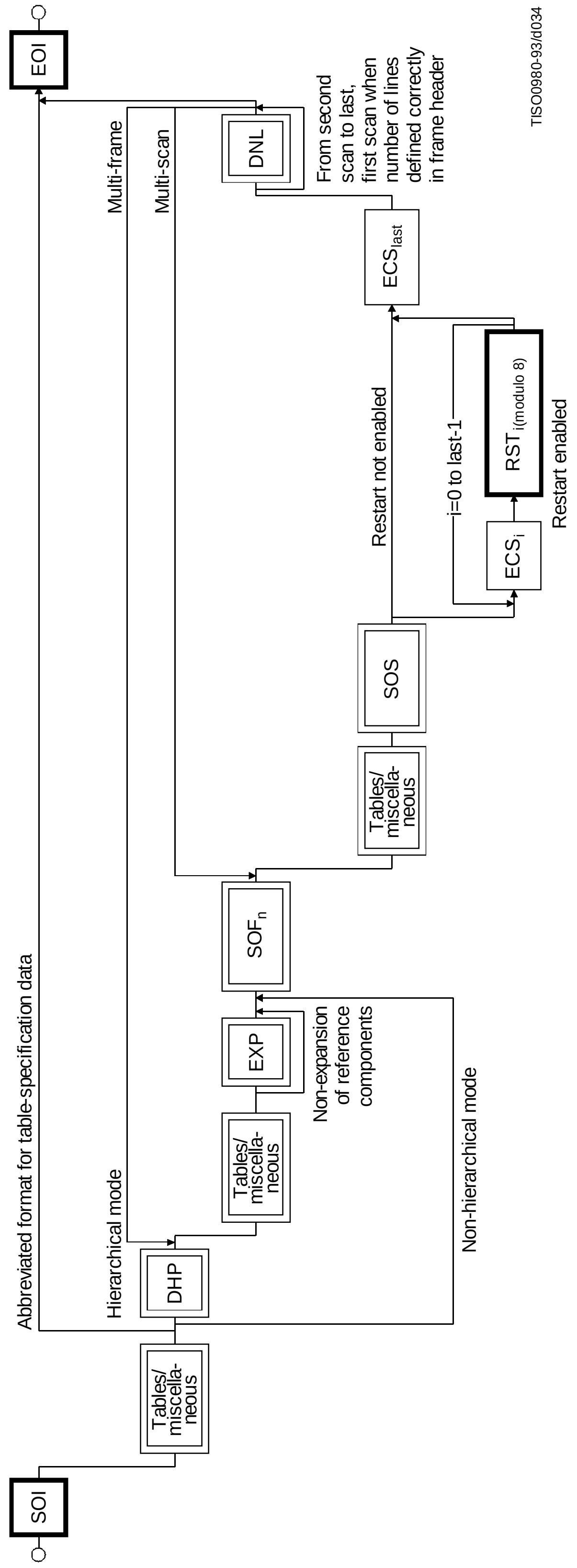


Figure B.16 – Flow of compressed data syntax

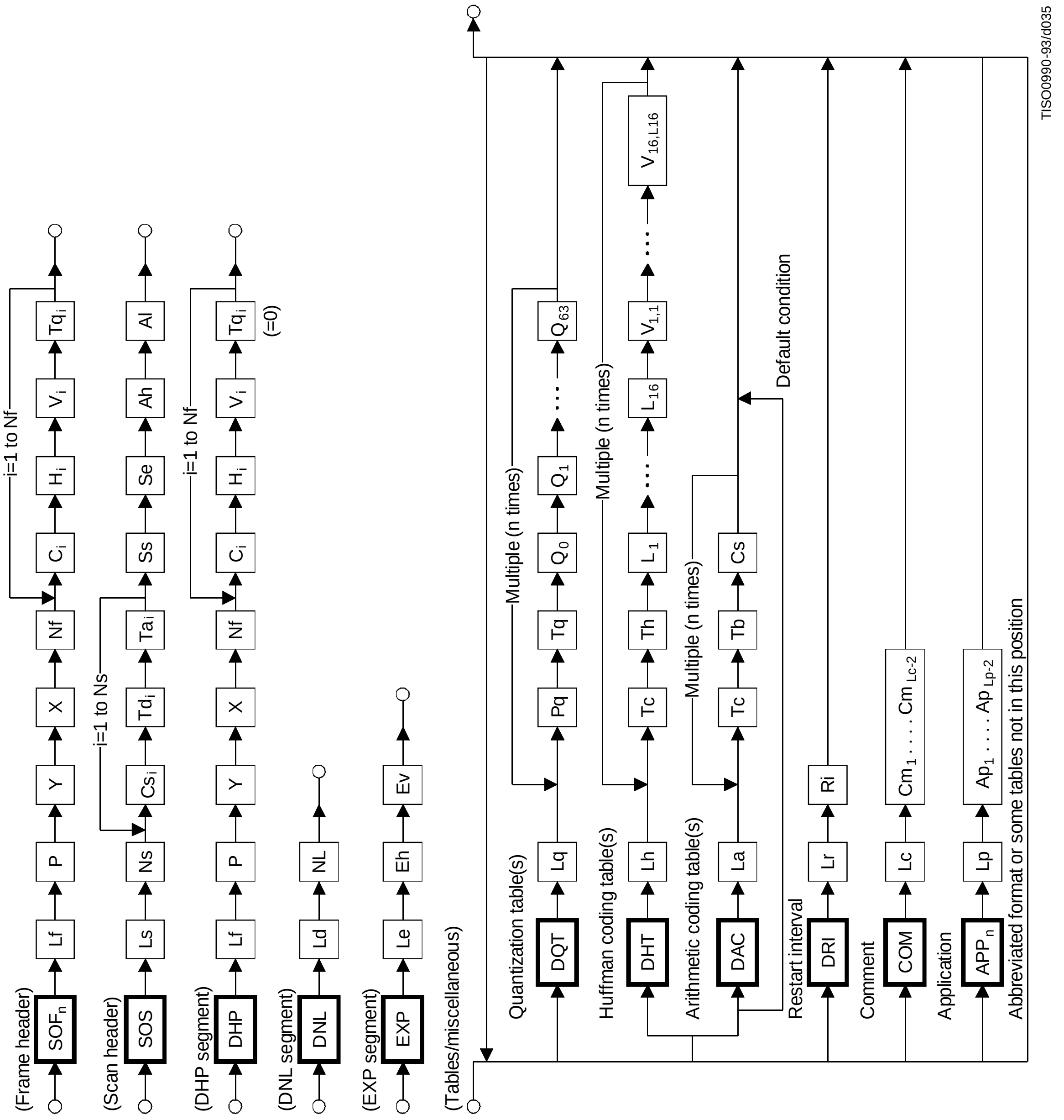


Figure B.17 – Flow of marker segment

Annex C

Huffman table specification

(This annex forms an integral part of this Recommendation | International Standard)

A Huffman coding procedure may be used for entropy coding in any of the coding processes. Coding models for Huffman encoding are defined in Annexes F, G, and H. In this Annex, the Huffman table specification is defined.

Huffman tables are specified in terms of a 16-byte list (BITS) giving the number of codes for each code length from 1 to 16. This is followed by a list of the 8-bit symbol values (HUFFVAL), each of which is assigned a Huffman code. The symbol values are placed in the list in order of increasing code length. Code lengths greater than 16 bits are not allowed. In addition, the codes shall be generated such that the all-1-bits code word of any length is reserved as a prefix for longer code words.

NOTE – The order of the symbol values within HUFFVAL is determined only by code length. Within a given code length the ordering of the symbol values is arbitrary.

This annex specifies the procedure by which the Huffman tables (of Huffman code words and their corresponding 8-bit symbol values) are derived from the two lists (BITS and HUFFVAL) in the interchange format. However, the way in which these lists are generated is not specified. The lists should be generated in a manner which is consistent with the rules for Huffman coding, and it shall observe the constraints discussed in the previous paragraph. Annex K contains an example of a procedure for generating lists of Huffman code lengths and values which are in accord with these rules.

NOTE – There is **no requirement** in this Specification that any encoder or decoder shall implement the procedures in precisely the manner specified by the flow charts in this annex. It is necessary only that an encoder or decoder implement the **function** specified in this annex. The sole criterion for an encoder or decoder to be considered in compliance with this Specification is that it satisfy the requirements given in clause 6 (for encoders) or clause 7 (for decoders), as determined by the compliance tests specified in Part 2.

C.1 Marker segments for Huffman table specification

The DHT marker identifies the start of Huffman table definitions within the compressed image data. B.2.4.2 specifies the syntax for Huffman table specification.

C.2 Conversion of Huffman table specifications to tables of codes and code lengths

Conversion of Huffman table specifications to tables of codes and code lengths uses three procedures. The first procedure (Figure C.1) generates a table of Huffman code sizes. The second procedure (Figure C.2) generates the Huffman codes from the table built in Figure C.1. The third procedure (Figure C.3) generates the Huffman codes in symbol value order.

Given a list BITS (1 to 16) containing the number of codes of each size, and a list HUFFVAL containing the symbol values to be associated with those codes as described above, two tables are generated. The HUFFSIZE table contains a list of code lengths; the HUFFCODE table contains the Huffman codes corresponding to those lengths.

Note that the variable LASTK is set to the index of the last entry in the table.

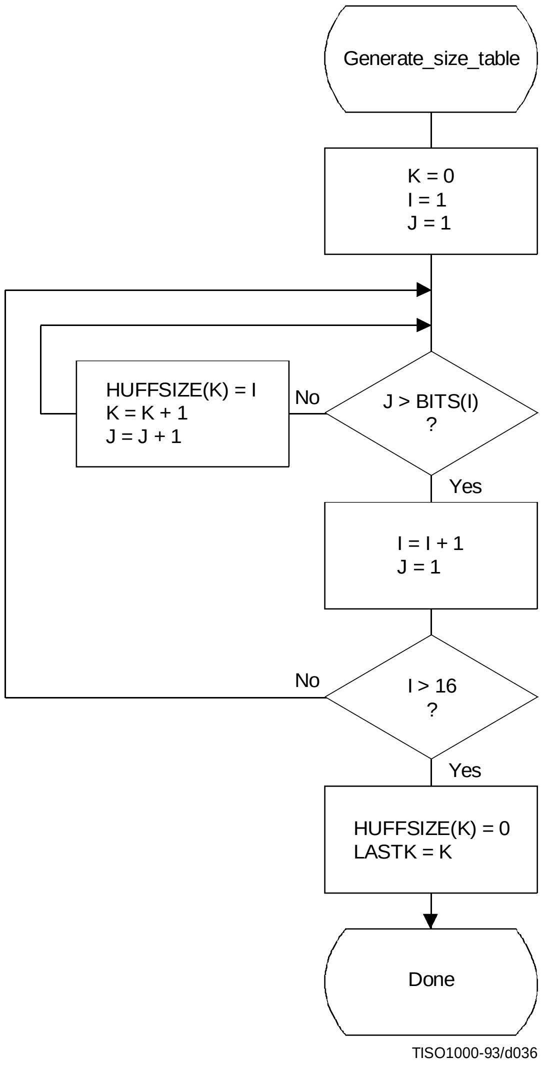


Figure C.1 – Generation of table of Huffman code sizes

A Huffman code table, HUFFCODE, containing a code for each size in HUFFSIZE is generated by the procedure in Figure C.2. The notation "SLL CODE 1" in Figure C.2 indicates a shift-left-logical of CODE by one bit position.

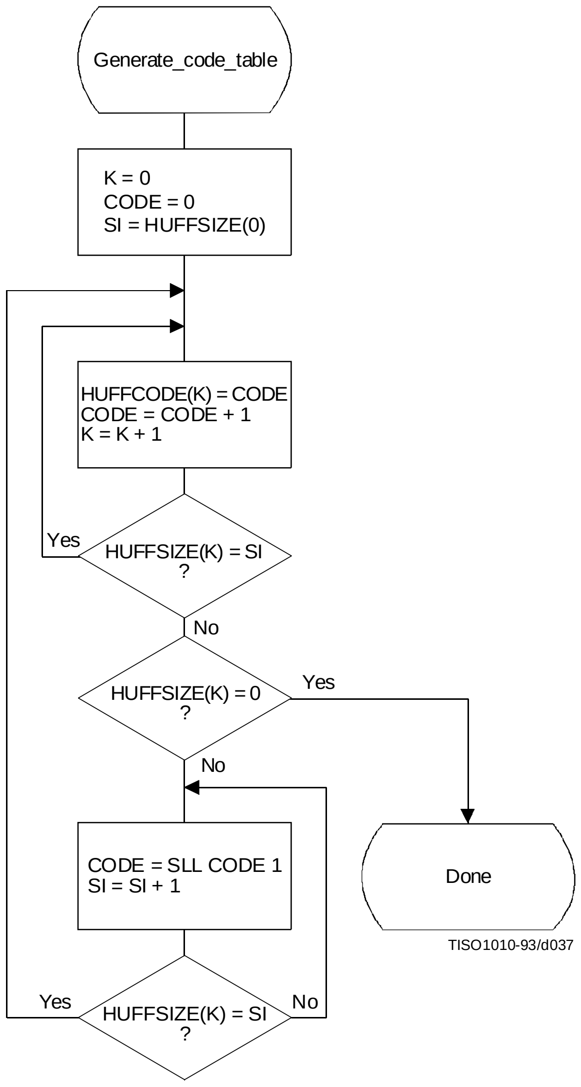


Figure C.2 – Generation of table of Huffman codes

Two tables, HUFFCODE and HUFFSIZE, have now been generated. The entries in the tables are ordered according to increasing Huffman code numeric value and length.

The encoding procedure code tables, EHUFCO and EHUFSI, are created by reordering the codes specified by HUFFCODE and HUFFSIZE according to the symbol values assigned to each code in HUFFVAL.

Figure C.3 illustrates this ordering procedure.

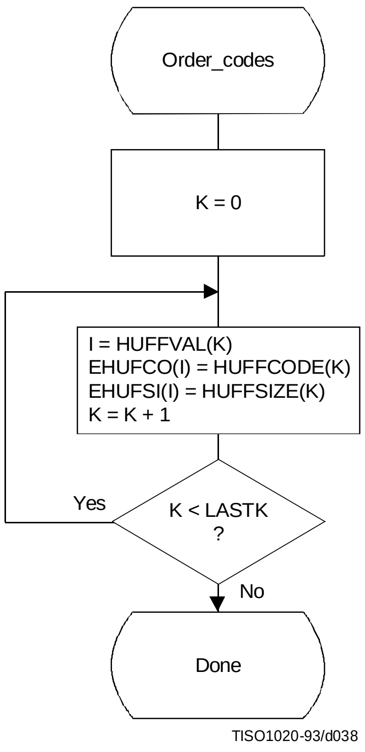


Figure C.3 – Ordering procedure for encoding procedure code tables

C.3 Bit ordering within bytes

The root of a Huffman code is placed toward the MSB (most-significant-bit) of the byte, and successive bits are placed in the direction MSB to LSB (least-significant-bit) of the byte. Remaining bits, if any, go into the next byte following the same rules.

Integers associated with Huffman codes are appended with the MSB adjacent to the LSB of the preceding Huffman code.

Annex D

Arithmetic coding

(This annex forms an integral part of this Recommendation | International Standard)

An adaptive binary arithmetic coding procedure may be used for entropy coding in any of the coding processes except the baseline sequential process. Coding models for adaptive binary arithmetic coding are defined in Annexes F, G, and H. In this annex the arithmetic encoding and decoding procedures used in those models are defined.

In K.4 a simple test example is given which should be helpful in determining if a given implementation is correct.

NOTE – There is **no requirement** in this Specification that any encoder or decoder shall implement the procedures in precisely the manner specified by the flow charts in this annex. It is necessary only that an encoder or decoder implement the **function** specified in this annex. The sole criterion for an encoder or decoder to be considered in compliance with this Specification is that it satisfy the requirements given in clause 6 (for encoders) or clause 7 (for decoders), as determined by the compliance tests specified in Part 2.

D.1 Arithmetic encoding procedures

Four arithmetic encoding procedures are required in a system with arithmetic coding (see Table D.1).

Table D.1 – Procedures for binary arithmetic encoding

Procedure	Purpose
Code_0(S)	Code a "0" binary decision with context-index S
Code_1(S)	Code a "1" binary decision with context-index S
Initenc	Initialize the encoder
Flush	Terminate entropy-coded segment

The "Code_0(S)"and "Code_1(S)" procedures code the 0-decision and 1-decision respectively; S is a context-index which identifies a particular conditional probability estimate used in coding the binary decision. The "Initenc" procedure initializes the arithmetic coding entropy encoder. The "Flush" procedure terminates the entropy-coded segment in preparation for the marker which follows.

D.1.1 Binary arithmetic encoding principles

The arithmetic coder encodes a series of binary symbols, zeros and ones, each symbol representing one possible result of a binary decision.

Each "binary decision" provides a choice between two alternatives. The binary decision might be between positive and negative signs, a magnitude being zero or nonzero, or a particular bit in a sequence of binary digits being zero or one.

The output bit stream (entropy-coded data segment) represents a binary fraction which increases in precision as bytes are appended by the encoding process.

D.1.1.1 Recursive interval subdivision

Recursive probability interval subdivision is the basis for the binary arithmetic encoding procedures. With each binary decision the current probability interval is subdivided into two sub-intervals, and the bit stream is modified (if necessary) so that it points to the base (the lower bound) of the probability sub-interval assigned to the symbol which occurred.

In the partitioning of the current probability interval into two sub-intervals, the sub-interval for the less probable symbol (LPS) and the sub-interval for the more probable symbol (MPS) are ordered such that usually the MPS sub-interval is closer to zero. Therefore, when the LPS is coded, the MPS sub-interval size is added to the bit stream. This coding convention requires that symbols be recognized as either MPS or LPS rather than 0 or 1. Consequently, the size of the LPS sub-interval and the sense of the MPS for each decision must be known in order to encode that decision.

The subdivision of the current probability interval would ideally require a multiplication of the interval by the probability estimate for the LPS. Because this subdivision is done approximately, it is possible for the LPS sub-interval to be larger than the MPS sub-interval. When that happens a “conditional exchange” interchanges the assignment of the sub-intervals such that the MPS is given the larger sub-interval.

Since the encoding procedure involves addition of binary fractions rather than concatenation of integer code words, the more probable binary decisions can sometimes be coded at a cost of much less than one bit per decision.

D.1.1.2 Conditioning of probability estimates

An adaptive binary arithmetic coder requires a statistical model – a model for selecting conditional probability estimates to be used in the coding of each binary decision. When a given binary decision probability estimate is dependent on a particular feature or features (the context) already coded, it is “conditioned” on that feature. The conditioning of probability estimates on previously coded decisions must be identical in encoder and decoder, and therefore can use only information known to both.

Each conditional probability estimate required by the statistical model is kept in a separate storage location or “bin” identified by a unique context-index S. The arithmetic coder is adaptive, which means that the probability estimates at each context-index are developed and maintained by the arithmetic coding system on the basis of prior coding decisions for that context-index.

D.1.2 Encoding conventions and approximations

The encoding procedures use fixed precision integer arithmetic and an integer representation of fractional values in which X’8000’ can be regarded as the decimal value 0.75. The probability interval, A, is kept in the integer range X’8000’ $\leq$ A $<$ X’10000’ by doubling it whenever its integer value falls below X’8000’. This is equivalent to keeping A in the decimal range $0.75 \leq A < 1.5$. This doubling procedure is called renormalization.

The code register, C, contains the trailing bits of the bit stream. C is also doubled each time A is doubled. Periodically – to keep C from overflowing – a byte of data is removed from the high order bits of the C-register and placed in the entropy-coded segment.

Carry-over into the entropy-coded segment is limited by delaying X’FF’ output bytes until the carry-over is resolved. Zero bytes are stuffed after each X’FF’ byte in the entropy-coded segment in order to avoid the accidental generation of markers in the entropy-coded segment.

Keeping A in the range $0.75 \leq A < 1.5$ allows a simple arithmetic approximation to be used in the probability interval subdivision. Normally, if the current estimate of the LPS probability for context-index S is Qe(S), precise calculation of the sub-intervals would require:

$Qe(S) \times A$	Probability sub-interval for the LPS;
$A - (Qe(S) \times A)$	Probability sub-interval for the MPS.

Because the decimal value of A is of order unity, these can be approximated by

$Qe(S)$	Probability sub-interval for the LPS;
$A - Qe(S)$	Probability sub-interval for the MPS.

Whenever the LPS is coded, the value of A – Qe(S) is added to the code register and the probability interval is reduced to Qe(S). Whenever the MPS is coded, the code register is left unchanged and the interval is reduced to A – Qe(S). The precision range required for A is then restored, if necessary, by renormalization of both A and C.

With the procedure described above, the approximations in the probability interval subdivision process can sometimes make the LPS sub-interval larger than the MPS sub-interval. If, for example, the value of Qe(S) is 0.5 and A is at the minimum allowed value of 0.75, the approximate scaling gives one-third of the probability interval to the MPS and two-thirds to the LPS. To avoid this size inversion, conditional exchange is used. The probability interval is subdivided using the simple approximation, but the MPS and LPS sub-interval assignments are exchanged whenever the LPS sub-interval is larger than the MPS sub-interval. This MPS/LPS conditional exchange can only occur when a renormalization will be needed.

Each binary decision uses a context. A context is the set of prior coding decisions which determine the context-index, S, identifying the probability estimate used in coding the decision.

Whenever a renormalization occurs, a probability estimation procedure is invoked which determines a new probability estimate for the context currently being coded. No explicit symbol counts are needed for the estimation. The relative probabilities of renormalization after coding of LPS and MPS provide, by means of a table-based probability estimation state machine, a direct estimate of the probabilities.

D.1.3 Encoder code register conventions

The flow charts in this annex assume the register structures for the encoder as shown in Table D.2.

Table D.2 – Encoder register connections

	MSB			LSB
C-register	0000cbbb,	bbbbbsss,	xxxxxxxx,	xxxxxxxx
A-register	00000000,	00000000,	aaaaaaaa,	aaaaaaaa

The "a" bits are the fractional bits in the A-register (the current probability interval value) and the "x" bits are the fractional bits in the code register. The "s" bits are optional spacer bits which provide useful constraints on carry-over, and the "b" bits indicate the bit positions from which the completed bytes of data are removed from the C-register. The "c" bit is a carry bit. Except at the time of initialization, bit 15 of the A-register is always set and bit 16 is always clear (the LSB is bit 0).

These register conventions illustrate one possible implementation. However, any register conventions which allow resolution of carry-over in the encoder and which produce the same entropy-coded segment may be used. The handling of carry-over and the byte stuffing following X'FF' will be described in a later part of this annex.

D.1.4 Code_1(S) and Code_0(S) procedures

When a given binary decision is coded, one of two possibilities occurs – either a 1-decision or a 0-decision is coded. Code_1(S) and Code_0(S) are shown in Figures D.1 and D.2. The Code_1(S) and Code_0(S) procedures use probability estimates with a context-index S. The context-index S is determined by the statistical model and is, in general, a function of the previous coding decisions; each value of S identifies a particular conditional probability estimate which is used in encoding the binary decision.

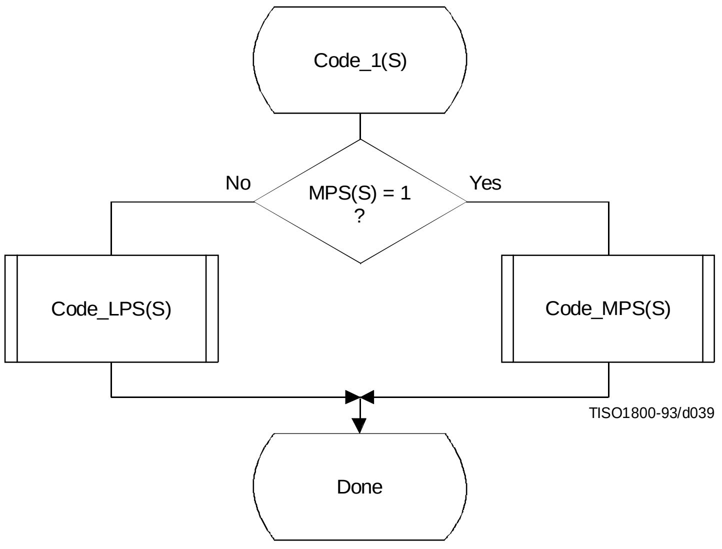


Figure D.1 – Code_1(S) procedure

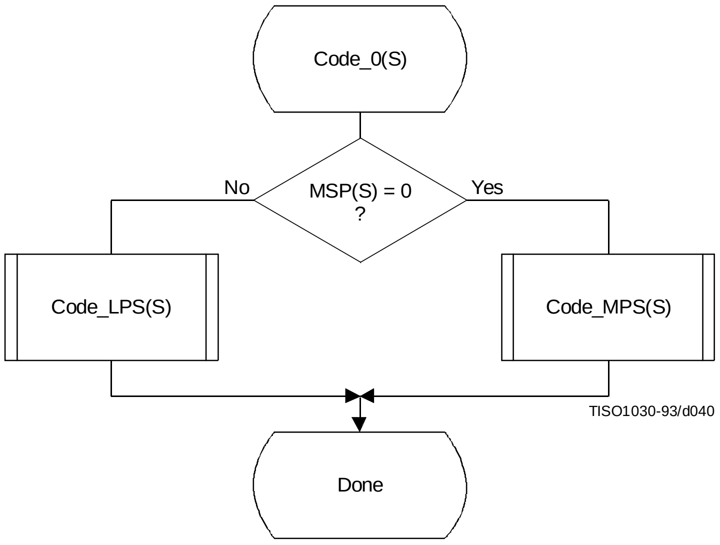


Figure D.2 – Code_0(S) procedure

The context-index S selects a storage location which contains Index(S), an index to the tables which make up the probability estimation state machine. When coding a binary decision, the symbol being coded is either the more probable symbol or the less probable symbol. Therefore, additional information is stored at each context-index identifying the sense of the more probable symbol, MPS(S).

For simplicity, the flow charts in this subclause assume that the context storage for each context-index S has an additional storage field for Qe(S) containing the value of Qe(Index(S)). If only the value of Index(S) and MPS(S) are stored, all references to Qe(S) should be replaced by Qe(Index(S)).

The Code_LPS(S) procedure normally consists of the addition of the MPS sub-interval A – Qe(S) to the bit stream and a scaling of the interval to the sub-interval, Qe(S). It is always followed by the procedures for obtaining a new LPS probability estimate (Estimate_Qe(S)_after_LPS) and renormalization (Renorm_e) (see Figure D.3).

However, in the event that the LPS sub-interval is larger than the MPS sub-interval, the conditional MPS/LPS exchange occurs and the MPS sub-interval is coded.

The Code_MPS(S) procedure normally reduces the size of the probability interval to the MPS sub-interval. However, if the LPS sub-interval is larger than the MPS sub-interval, the conditional exchange occurs and the LPS sub-interval is coded instead. Note that conditional exchange cannot occur unless the procedures for obtaining a new LPS probability estimate (Estimate_Qe(S)_after_MPS) and renormalization (Renorm_e) are required after the coding of the symbol (see Figure D.4).

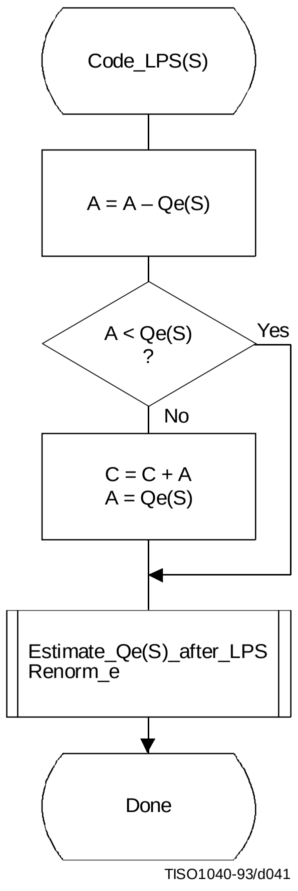


Figure D.3 – Code_LPS(S) procedure with conditional MPS/LPS exchange

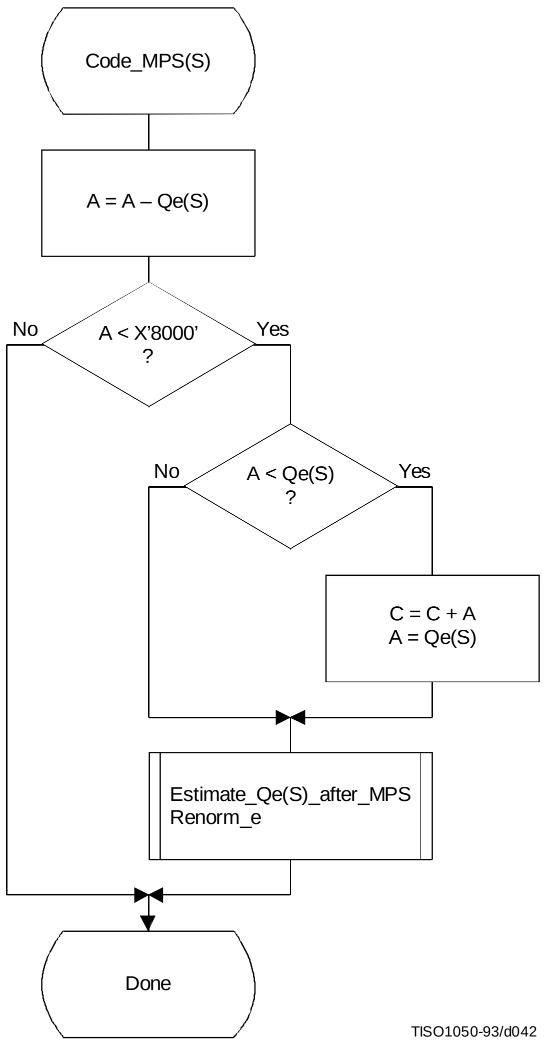


Figure D.4 – Code_MPS(S) procedure with conditional MPS/LPS exchange

D.1.5 Probability estimation in the encoder

D.1.5.1 Probability estimation state machine

The probability estimation state machine consists of a number of sequences of probability estimates. These sequences are interlinked in a manner which provides probability estimates based on approximate symbol counts derived from the arithmetic coder renormalization. Some of these sequences are used during the initial "learning" stages of probability estimation; the rest are used for "steady state" estimation.

Each entry in the probability estimation state machine is assigned an index, and each index has associated with it a Qe value and two Next_Index values. The Next_Index_MPS gives the index to the new probability estimate after an MPS renormalization; the Next_Index_LPS gives the index to the new probability estimate after an LPS renormalization. Note that both the index to the estimation state machine and the sense of the MPS are kept for each context-index S. The sense of the MPS is changed whenever the entry in the Switch_MPS is one.

The probability estimation state machine is given in Table D.3. Initialization of the arithmetic coder is always with an MPS sense of zero and a Qe index of zero in Table D.3.

The Qe values listed in Table D.3 are expressed as hexadecimal integers. To approximately convert the 15-bit integer representation of Qe to a decimal probability, divide the Qe values by (4/3) × (X'8000').

Table D.3 – Qe values and probability estimation state machine

Index	Qe	Next_ Index		Switch	Index	Qe	Next_ Index		Switch
	_Value	_LPS	_MPS	_MPS		_Value	_LPS	_MPS	_MPS
0	X’5A1D’	1	1	1	57	X’01A4’	55	58	0
1	X’2586’	14	2	0	58	X’0160’	56	59	0
2	X’1114’	16	3	0	59	X’0125’	57	60	0
3	X’080B’	18	4	0	60	X’00F6’	58	61	0
4	X’03D8’	20	5	0	61	X’00CB’	59	62	0
5	X’01DA’	23	6	0	62	X’00AB’	61	63	0
6	X’00E5’	25	7	0	63	X’008F’	61	32	0
7	X’006F’	28	8	0	64	X’5B12’	65	65	1
8	X’0036’	30	9	0	65	X’4D04’	80	66	0
9	X’001A’	33	10	0	66	X’412C’	81	67	0
10	X’000D’	35	11	0	67	X’37D8’	82	68	0
11	X’0006’	9	12	0	68	X’2FE8’	83	69	0
12	X’0003’	10	13	0	69	X’293C’	84	70	0
13	X’0001’	12	13	0	70	X’2379’	86	71	0
14	X’5A7F’	15	15	1	71	X’1EDF’	87	72	0
15	X’3F25’	36	16	0	72	X’1AA9’	87	73	0
16	X’2CF2’	38	17	0	73	X’174E’	72	74	0
17	X’207C’	39	18	0	74	X’1424’	72	75	0
18	X’17B9’	40	19	0	75	X’119C’	74	76	0
19	X’1182’	42	20	0	76	X’0F6B’	74	77	0
20	X’0CEF’	43	21	0	77	X’0D51’	75	78	0
21	X’09A1’	45	22	0	78	X’0BB6’	77	79	0
22	X’072F’	46	23	0	79	X’0A40’	77	48	0
23	X’055C’	48	24	0	80	X’5832’	80	81	1
24	X’0406’	49	25	0	81	X’4D1C’	88	82	0
25	X’0303’	51	26	0	82	X’438E’	89	83	0
26	X’0240’	52	27	0	83	X’3BDD’	90	84	0
27	X’01B1’	54	28	0	84	X’34EE’	91	85	0
28	X’0144’	56	29	0	85	X’2EAE’	92	86	0
29	X’00F5’	57	30	0	86	X’299A’	93	87	0
30	X’00B7’	59	31	0	87	X’2516’	86	71	0
31	X’008A’	60	32	0	88	X’5570’	88	89	1
32	X’0068’	62	33	0	89	X’4CA9’	95	90	0
33	X’004E’	63	34	0	90	X’44D9’	96	91	0
34	X’003B’	32	35	0	91	X’3E22’	97	92	0
35	X’002C’	33	9	0	92	X’3824’	99	93	0
36	X’5AE1’	37	37	1	93	X’32B4’	99	94	0
37	X’484C’	64	38	0	94	X’2E17’	93	86	0
38	X’3A0D’	65	39	0	95	X’56A8’	95	96	1
39	X’2EF1’	67	40	0	96	X’4F46’	101	97	0
40	X’261F’	68	41	0	97	X’47E5’	102	98	0
41	X’1F33’	69	42	0	98	X’41CF’	103	99	0
42	X’19A8’	70	43	0	99	X’3C3D’	104	100	0
43	X’1518’	72	44	0	100	X’375E’	99	93	0
44	X’1177’	73	45	0	101	X’5231’	105	102	0
45	X’0E74’	74	46	0	102	X’4C0F’	106	103	0
46	X’0BFB’	75	47	0	103	X’4639’	107	104	0
47	X’09F8’	77	48	0	104	X’415E’	103	99	0
48	X’0861’	78	49	0	105	X’5627’	105	106	1
49	X’0706’	79	50	0	106	X’50E7’	108	107	0
50	X’05CD’	48	51	0	107	X’4B85’	109	103	0
51	X’04DE’	50	52	0	108	X’5597’	110	109	0
52	X’040F’	50	53	0	109	X’504F’	111	107	0
53	X’0363’	51	54	0	110	X’5A10’	110	111	1
54	X’02D4’	52	55	0	111	X’5522’	112	109	0
55	X’025C’	53	56	0	112	X’59EB’	112	111	1
56	X’01F8’	54	57	0					

D.1.5.2 Renormalization driven estimation

The change in state in Table D.3 occurs only when the arithmetic coder interval register is renormalized. This must always be done after coding an LPS, and whenever the probability interval register is less than X'8000' (0.75 in decimal notation) after coding an MPS.

When the LPS renormalization is required, Next_Index_LPS gives the new index for the LPS probability estimate. When the MPS renormalization is required, Next_Index_MPS gives the new index for the LPS probability estimate. If Switch_MPS is 1 for the old index, the MPS symbol sense must be inverted after an LPS.

D.1.5.3 Estimation following renormalization after MPS

The procedure for estimating the probability on the MPS renormalization path is given in Figure D.5. Index(S) is part of the information stored for context-index S. The new value of Index(S) is obtained from Table D.3 from the column labeled Next_Index_MPS, as that is the next index after an MPS renormalization. This next index is stored as the new value of Index(S) in the context storage at context-index S, and the value of Qe at this new Index(S) becomes the new Qe(S). MPS(S) does not change.

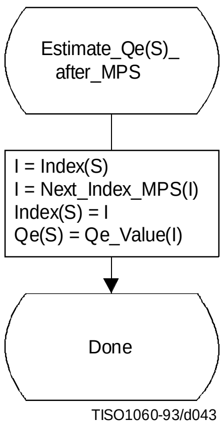


Figure D.5 – Probability estimation on MPS renormalization path

D.1.5.4 Estimation following renormalization after LPS

The procedure for estimating the probability on the LPS renormalization path is shown in Figure D.6. The procedure is similar to that of Figure D.5 except that when Switch_MPS(I) is 1, the sense of MPS(S) must be inverted.

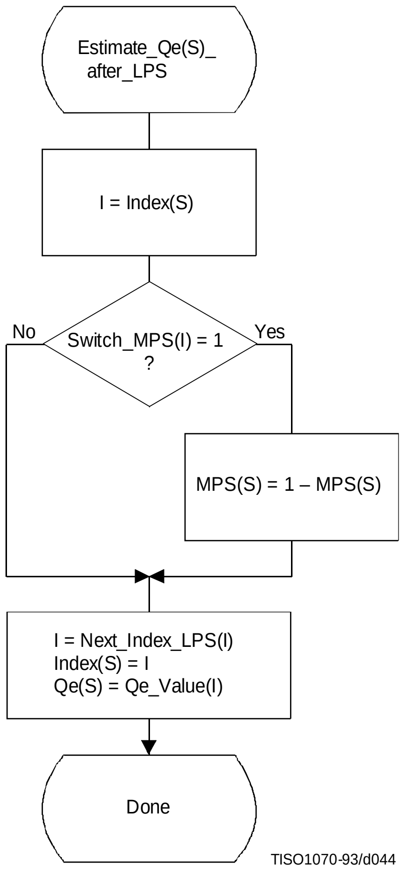


Figure D.6 – Probability estimation on LPS renormalization path

D.1.6 Renormalization in the encoder

The Renorm_e procedure for the encoder renormalization is shown in Figure D.7. Both the probability interval register A and the code register C are shifted, one bit at a time. The number of shifts is counted in the counter CT; when CT is zero, a byte of compressed data is removed from C by the procedure Byte_out and CT is reset to 8. Renormalization continues until A is no longer less than X'8000'.

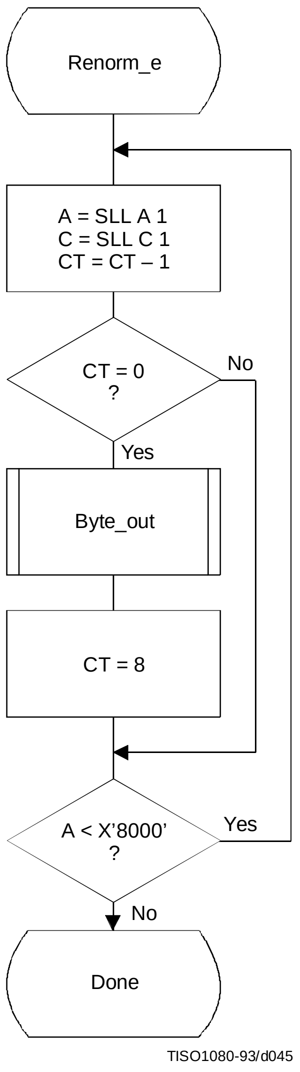


Figure D.7 – Encoder renormalization procedure

The Byte_out procedure used in Renorm_e is shown in Figure D.8. This procedure uses byte-stuffing procedures which prevent accidental generation of markers by the arithmetic encoding procedures. It also includes an example of a procedure for resolving carry-over. For simplicity of exposition, the buffer holding the entropy-coded segment is assumed to be large enough to contain the entire segment.

In Figure D.8 BP is the entropy-coded segment pointer and B is the compressed data byte pointed to by BP. T in Byte_out is a temporary variable which is used to hold the output byte and carry bit. ST is the stack counter which is used to count X’FF’ output bytes until any carry-over through the X’FF’ sequence has been resolved. The value of ST rarely exceeds 3. However, since the upper limit for the value of ST is bounded only by the total entropy-coded segment size, a precision of 32 bits is recommended for ST.

Since large values of ST represent a latent output of compressed data, the following procedure may be needed in high speed synchronous encoding systems for handling the burst of output data which occurs when the carry is resolved.

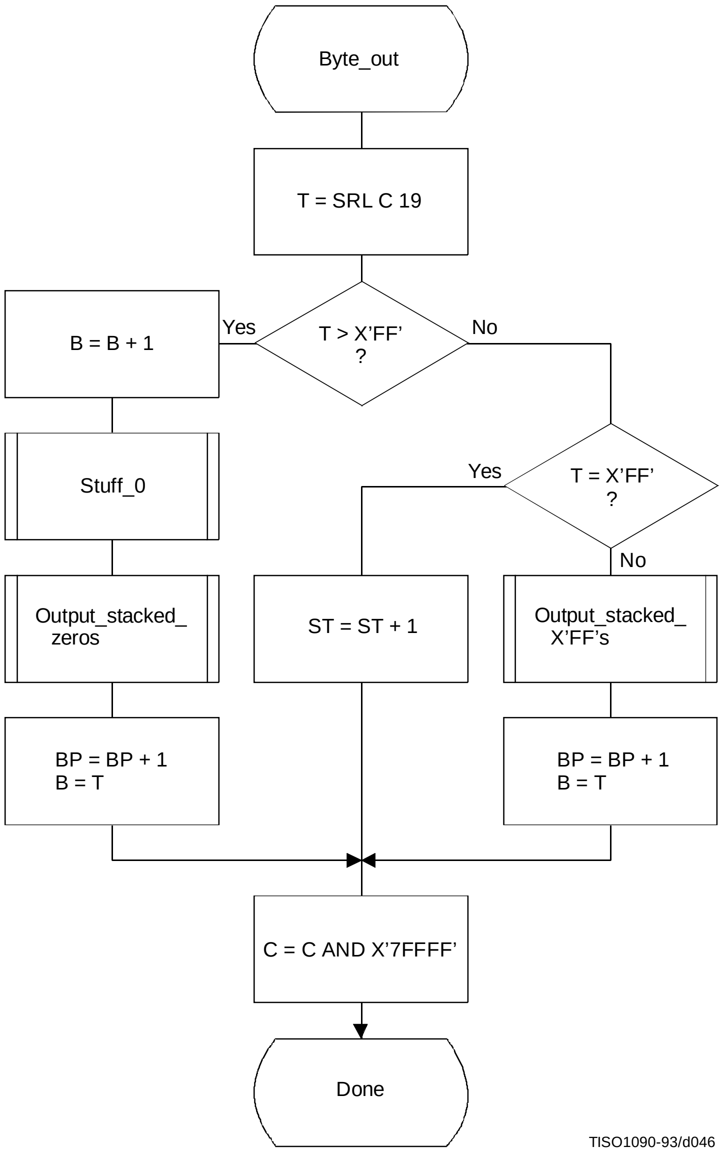


Figure D.8 – Byte_out procedure for encoder

When the stack count reaches an upper bound determined by output channel capacity, the stack is emptied and the stacked X'FF' bytes (and stuffed zero bytes) are added to the compressed data before the carry-over is resolved. If a carry-over then occurs, the carry is added to the final stuffed zero, thereby converting the final X'FF00' sequence to the X'FF01' temporary private marker. The entropy-coded segment must then be post-processed to resolve the carry-over and remove the temporary marker code. For any reasonable bound on ST this post processing is very unlikely.

Referring to Figure D.8, the shift of the code register by 19 bits aligns the output bits with the low order bits of T. The first test then determines if a carry-over has occurred. If so, the carry must be added to the previous output byte before advancing the segment pointer BP. The Stuff_0 procedure stuffs a zero byte whenever the addition of the carry to the data already in the entropy-coded segments creates a X'FF' byte. Any stacked output bytes – converted to zeros by the carry-over – are then placed in the entropy-coded segment. Note that when the output byte is later transferred from T to the entropy-coded segment (to byte B), the carry bit is ignored if it is set.

If a carry has not occurred, the output byte is tested to see if it is X’FF’. If so, the stack count ST is incremented, as the output must be delayed until the carry-over is resolved. If not, the carry-over has been resolved, and any stacked X’FF’ bytes must then be placed in the entropy-coded segment. Note that a zero byte is stuffed following each X’FF’.

The procedures used by Byte_out are defined in Figures D.9 through D.11.

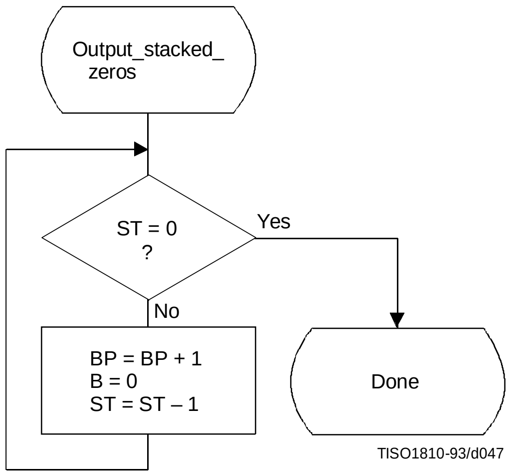


Figure D.9 – Output_stacked_zeros procedure for encoder

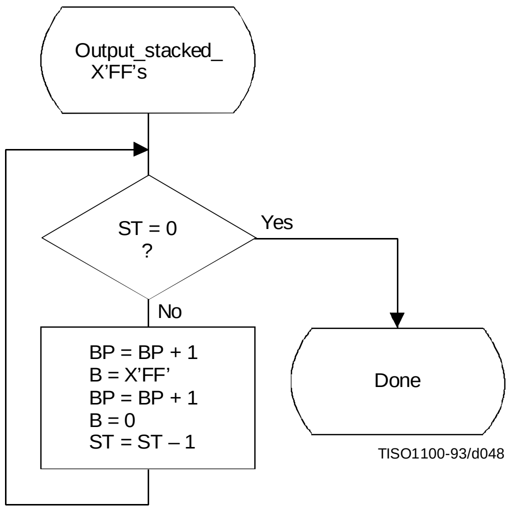


Figure D.10 – Output_stacked_X’FF’s procedure for encoder

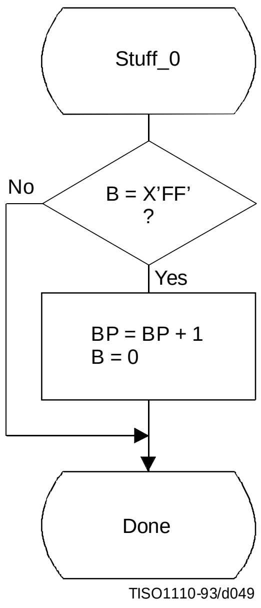


Figure D.11 – Stuff_0 procedure for encoder

D.1.7 Initialization of the encoder

The Initenc procedure is used to start the arithmetic coder. The basic steps are shown in Figure D.12.

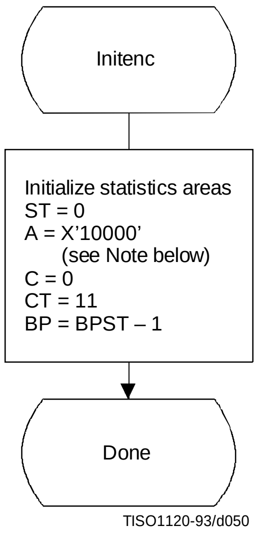


Figure D.12 – Initialization of the encoder

The probability estimation tables are defined by Table D.3. The statistics areas are initialized to an MPS sense of 0 and a Qe index of zero as defined by Table D.3. The stack count (ST) is cleared, the code register (C) is cleared, and the interval register is set to X'10000'. The counter (CT) is set to 11, reflecting the fact that when A is initialized to X'10000' three spacer bits plus eight output bits in C must be filled before the first byte is removed. Note that BP is initialized to point to the byte before the start of the entropy-coded segment (which is at BPST). Note also that the statistics areas are initialized for all values of context-index S to MPS(S) = 0 and Index(S) = 0.

NOTE – Although the probability interval is initialized to X'10000' in both Initenc and Initdec, the precision of the probability interval register can still be limited to 16 bits. When the precision of the interval register is 16 bits, it is initialized to zero.

D.1.8 Termination of encoding

The Flush procedure is used to terminate the arithmetic encoding procedures and prepare the entropy-coded segment for the addition of the X'FF' prefix of the marker which follows the arithmetically coded data. Figure D.13 shows this flush procedure. The first step in the procedure is to set as many low order bits of the code register to zero as possible without pointing outside of the final interval. Then, the output byte is aligned by shifting it left by CT bits; Byte_out then removes it from C. C is then shifted left by 8 bits to align the second output byte and Byte_out is used a second time. The remaining low order bits in C are guaranteed to be zero, and these trailing zero bits shall not be written to the entropy-coded segment.

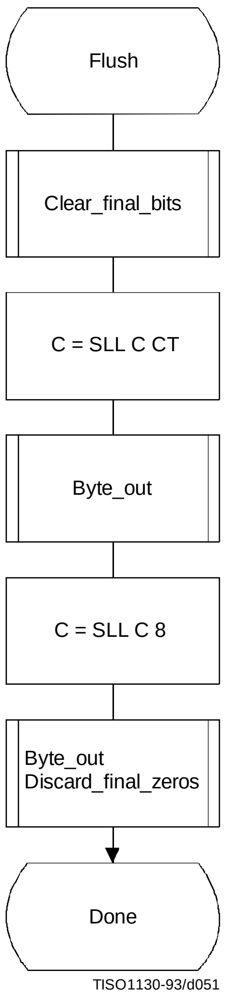


Figure D.13 – Flush procedure

Any trailing zero bytes already written to the entropy-coded segment and not preceded by a X’FF’ may, optionally, be discarded. This is done in the Discard_final_zeros procedure. Stuffed zero bytes shall not be discarded.

Entropy coded segments are always followed by a marker. For this reason, the final zero bits needed to complete decoding shall not be included in the entropy coded segment. Instead, when the decoder encounters a marker, zero bits shall be supplied to the decoding procedure until decoding is complete. This convention guarantees that when a DNL marker is used, the decoder will intercept it in time to correctly terminate the decoding procedure.

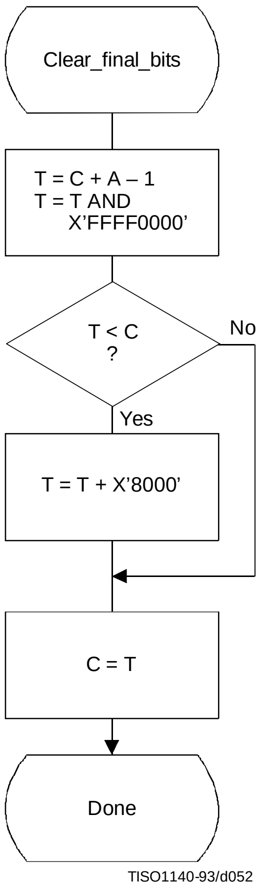


Figure D.14 – Clear_final_bits procedure in Flush

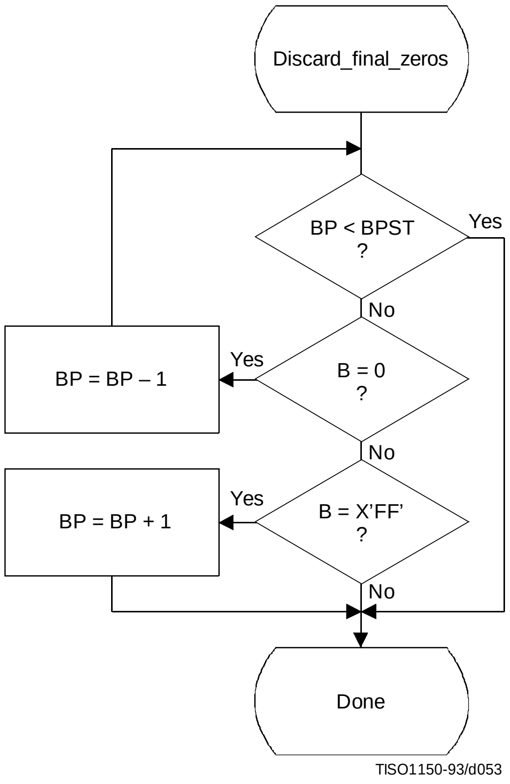


Figure D.15 – Discard_final_zeros procedure in Flush

D.2 Arithmetic decoding procedures

Two arithmetic decoding procedures are used for arithmetic decoding (see Table D.4).

The "Decode(S)" procedure decodes the binary decision for a given context-index S and returns a value of either 0 or 1. It is the inverse of the "Code_0(S)" and "Code_1(S)" procedures described in D.1. "Initdec" initializes the arithmetic coding entropy decoder.

Table D.4 – Procedures for binary arithmetic decoding

Procedure	Purpose
Decode(S)	Decode a binary decision with context-index S
Initdec	Initialize the decoder

D.2.1 Binary arithmetic decoding principles

The probability interval subdivision and sub-interval ordering defined for the arithmetic encoding procedures also apply to the arithmetic decoding procedures.

Since the bit stream always points within the current probability interval, the decoding process is a matter of determining, for each decision, which sub-interval is pointed to by the bit stream. This is done recursively, using the same probability interval sub-division process as in the encoder. Each time a decision is decoded, the decoder subtracts from the bit stream any interval the encoder added to the bit stream. Therefore, the code register in the decoder is a pointer into the current probability interval relative to the base of the interval.

If the size of the sub-interval allocated to the LPS is larger than the sub-interval allocated to the MPS, the encoder invokes the conditional exchange procedure. When the interval sizes are inverted in the decoder, the sense of the symbol decoded must be inverted.

D.2.2 Decoding conventions and approximations

The approximations and integer arithmetic defined for the probability interval subdivision in the encoder must also be used in the decoder. However, where the encoder would have added to the code register, the decoder subtracts from the code register.

D.2.3 Decoder code register conventions

The flow charts given in this section assume the register structures for the decoder as shown in Table D.5:

Table D.5 – Decoder register conventions

	MSB	LSB
Cx register	xxxxxxxx,	xxxxxxxx
C-low	bbbbbbbb,	00000000
A-register	aaaaaaaa,	aaaaaaaa

Cx and C-low can be regarded as one 32-bit C-register, in that renormalization of C shifts a bit of new data from bit 15 of C-low to bit 0 of Cx. However, the decoding comparisons use Cx alone. New data are inserted into the "b" bits of C-low one byte at a time.

NOTE – The comparisons shown in the various procedures use arithmetic comparisons, and therefore assume precisions greater than 16 bits for the variables. Unsigned (logical) comparisons should be used in 16-bit precision implementations.

D.2.4 The decode procedure

The decoder decodes one binary decision at a time. After decoding the decision, the decoder subtracts any amount from the code register that the encoder added. The amount left in the code register is the offset from the base of the current probability interval to the sub-interval allocated to the binary decisions not yet decoded. In the first test in the decode procedure shown in Figure D.16 the code register is compared to the size of the MPS sub-interval. Unless a conditional exchange is needed, this test determines whether the MPS or LPS for context-index S is decoded. Note that the LPS for context-index S is given by 1 – MPS(S).

When a renormalization is needed, the MPS/LPS conditional exchange may also be needed. For the LPS path, the conditional exchange procedure is shown in Figure D.17. Note that the probability estimation in the decoder is identical to the probability estimation in the encoder (Figures D.5 and D.6).

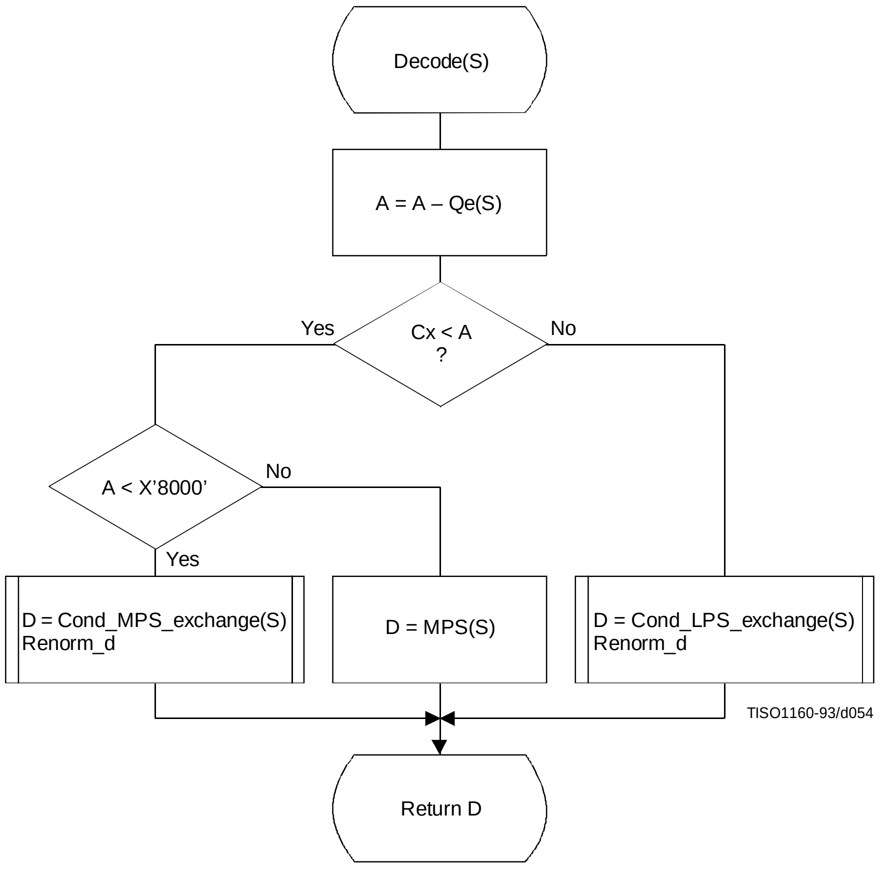


Figure D.16 – Decode(S) procedure

For the MPS path of the decoder the conditional exchange procedure is given in Figure D.18.

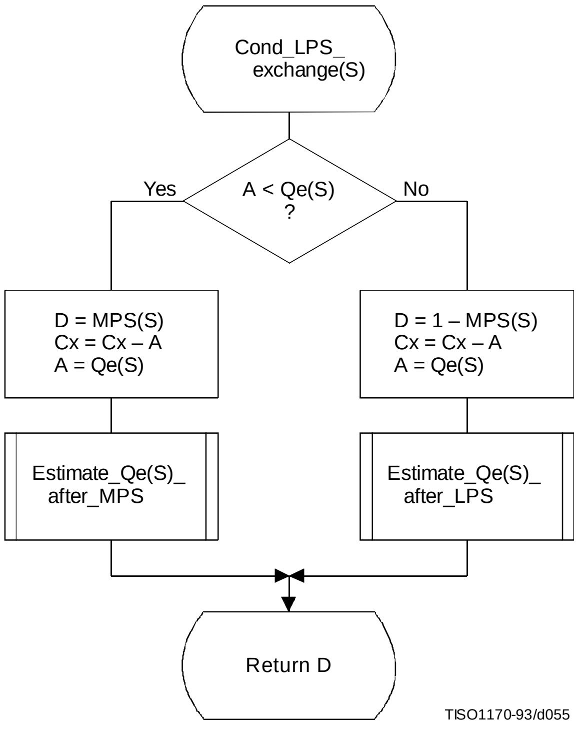


Figure D.17 – Decoder LPS path conditional exchange procedure

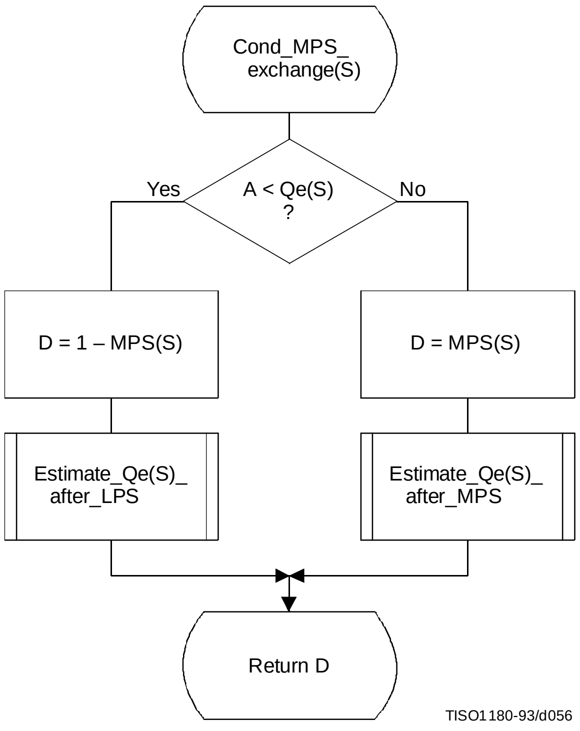


Figure D.18 – Decoder MPS path conditional exchange procedure

D.2.5 Probability estimation in the decoder

The procedures defined for obtaining a new LPS probability estimate in the encoder are also used in the decoder.

D.2.6 Renormalization in the decoder

The Renorm_d procedure for the decoder renormalization is shown in Figure D.19. CT is a counter which keeps track of the number of compressed bits in the C-low section of the C-register. When CT is zero, a new byte is inserted into C-low by the procedure Byte_in and CT is reset to 8.

Both the probability interval register A and the code register C are shifted, one bit at a time, until A is no longer less than X’8000’.

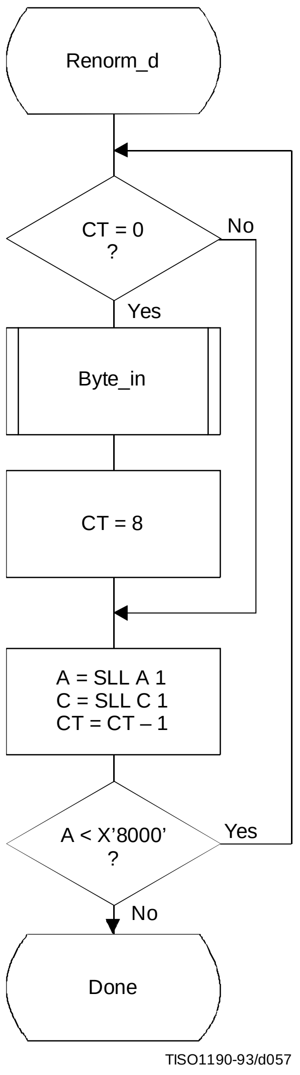


Figure D.19 – Decoder renormalization procedure

The Byte_in procedure used in Renorm_d is shown in Figure D.20. This procedure fetches one byte of data, compensating for the stuffed zero byte which follows any X'FF' byte. It also detects the marker which must follow the entropy-coded segment. The C-register in this procedure is the concatenation of the Cx and C-low registers. For simplicity of exposition, the buffer holding the entropy-coded segment is assumed to be large enough to contain the entire segment.

B is the byte pointed to by the entropy-coded segment pointer BP. BP is first incremented. If the new value of B is not a X'FF', it is inserted into the high order 8 bits of C-low.

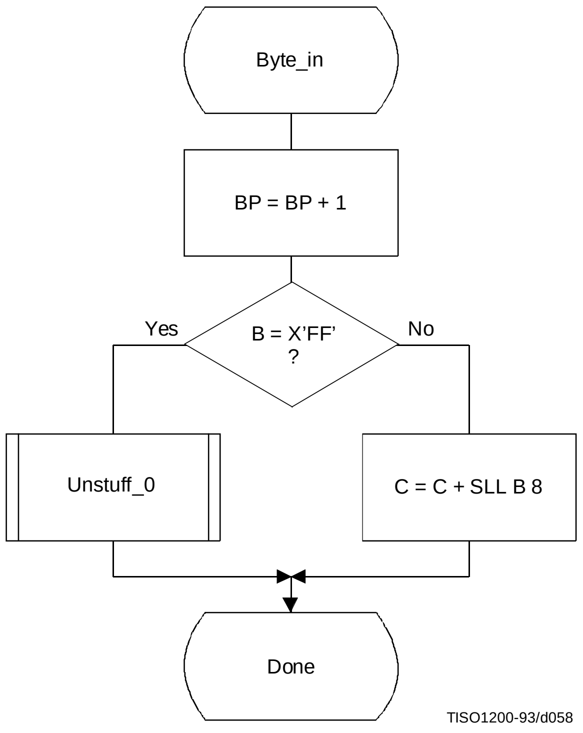


Figure D.20 – Byte_in procedure for decoder

The Unstuff_0 procedure is shown in Figure D.21. If the new value of B is X’FF’, BP is incremented to point to the next byte and this next B is tested to see if it is zero. If so, B contains a stuffed byte which must be skipped. The zero B is ignored, and the X’FF’ B value which preceded it is inserted in the C-register.

If the value of B after a X’FF’ byte is not zero, then a marker has been detected. The marker is interpreted as required and the entropy-coded segment pointer is adjusted (“Adjust BP” in Figure D.21) so that 0-bytes will be fed to the decoder until decoding is complete. One way of accomplishing this is to point BP to the byte preceding the marker which follows the entropy-coded segment.

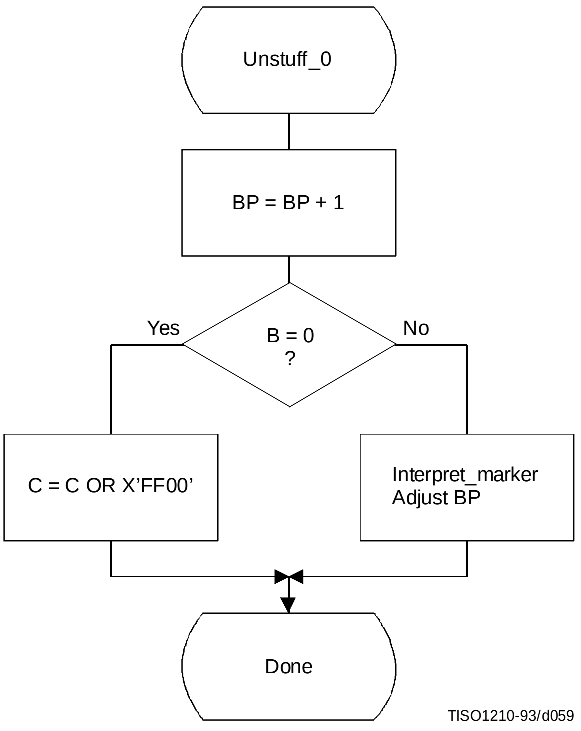


Figure D.21 – Unstuff_0 procedure for decoder

D.2.7 Initialization of the decoder

The Initdec procedure is used to start the arithmetic decoder. The basic steps are shown in Figure D.22.

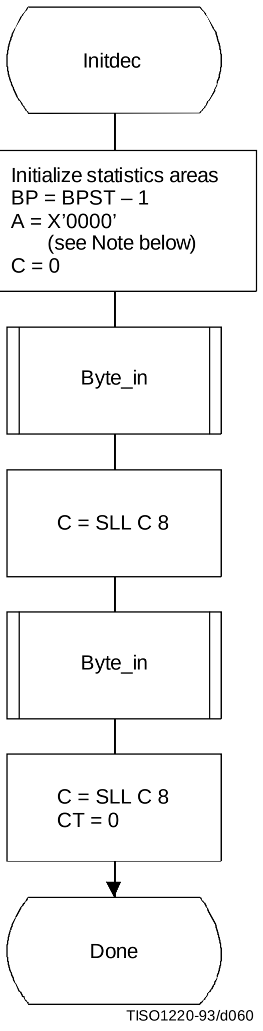


Figure D.22 – Initialization of the decoder

The estimation tables are defined by Table D.3. The statistics areas are initialized to an MPS sense of 0 and a Qe index of zero as defined by Table D.3. BP, the pointer to the entropy-coded segment, is then initialized to point to the byte before the start of the entropy-coded segment at BPST, and the interval register is set to the same starting value as in the encoder. The first byte of compressed data is fetched and shifted into Cx. The second byte is then fetched and shifted into Cx. The count is set to zero, so that a new byte of data will be fetched by Renorm_d.

NOTE – Although the probability interval is initialized to X'10000' in both Initenc and Initdec, the precision of the probability interval register can still be limited to 16 bits. When the precision of the interval register is 16 bits, it is initialized to zero.

D.3 Bit ordering within bytes

The arithmetically encoded entropy-coded segment is an integer of variable length. Therefore, the ordering of bytes and the bit ordering within bytes is the same as for parameters (see B.1.1.1).

Annex E

Encoder and decoder control procedures

(This annex forms an integral part of this Recommendation | International Standard)

This annex describes the encoder and decoder control procedures for the sequential, progressive, and lossless modes of operation.

The encoding and decoding control procedures for the hierarchical processes are specified in Annex J.

NOTES

1 There is **no requirement** in this Specification that any encoder or decoder shall implement the procedures in precisely the manner specified by the flow charts in this annex. It is necessary only that an encoder or decoder implement the **function** specified in this annex. The sole criterion for an encoder or decoder to be considered in compliance with this Specification is that it satisfy the requirements given in clause 6 (for encoders) or clause 7 (for decoders), as determined by the compliance tests specified in Part 2.

2 Implementation-specific setup steps are not indicated in this annex and may be necessary.

E.1 Encoder control procedures

E.1.1 Control procedure for encoding an image

The encoder control procedure for encoding an image is shown in Figure E.1.

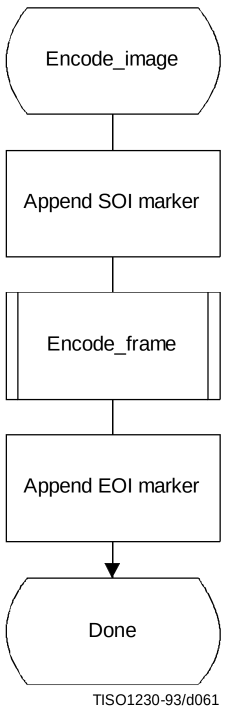


Figure E.1 – Control procedure for encoding an image

E.1.2 Control procedure for encoding a frame

In all cases where markers are appended to the compressed data, optional X'FF' fill bytes may precede the marker.

The control procedure for encoding a frame is oriented around the scans in the frame. The frame header is first appended, and then the scans are coded. Table specifications and other marker segments may precede the SOF_n marker, as indicated by [tables/miscellaneous] in Figure E.2.

Figure E.2 shows the encoding process frame control procedure.

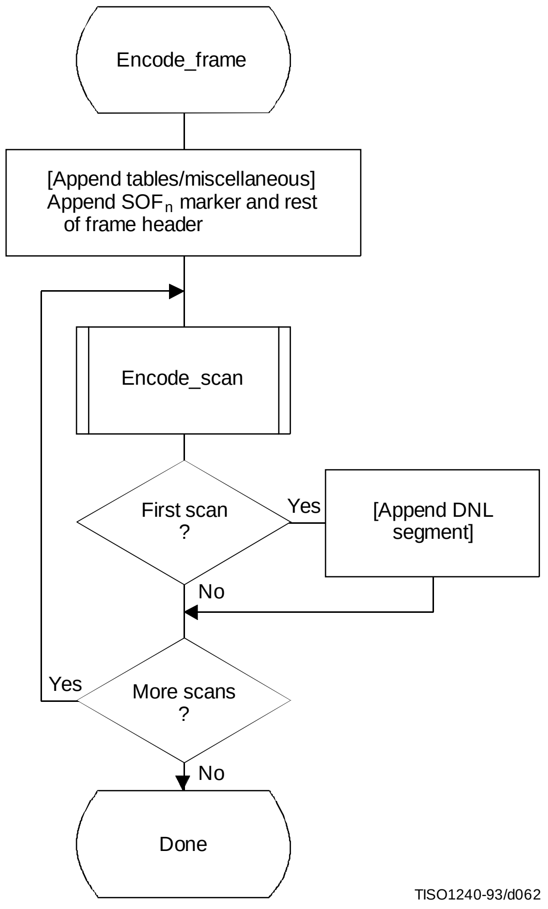


Figure E.2 – Control procedure for encoding a frame

E.1.3 Control procedure for encoding a scan

A scan consists of a single pass through the data of each component in the scan. Table specifications and other marker segments may precede the SOS marker. If more than one component is coded in the scan, the data are interleaved. If restart is enabled, the data are segmented into restart intervals. If restart is enabled, a RST_m marker is placed in the coded data between restart intervals. If restart is disabled, the control procedure is the same, except that the entire scan contains a single restart interval. The compressed image data generated by a scan is always followed by a marker, either the EOI marker or the marker of the next marker segment.

Figure E.3 shows the encoding process scan control procedure. The loop is terminated when the encoding process has coded the number of restart intervals which make up the scan. "m" is the restart interval modulo counter needed for the RST_m marker. The modulo arithmetic for this counter is shown after the "Append RST_m marker" procedure.

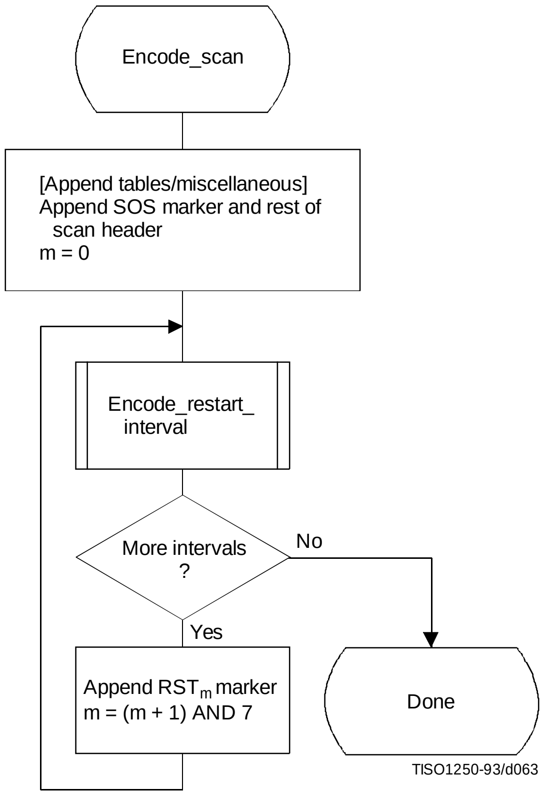


Figure E.3 – Control procedure for encoding a scan

E.1.4 Control procedure for encoding a restart interval

Figure E.4 shows the encoding process control procedure for a restart interval. The loop is terminated either when the encoding process has coded the number of minimum coded units (MCU) in the restart interval or when it has completed the image scan.

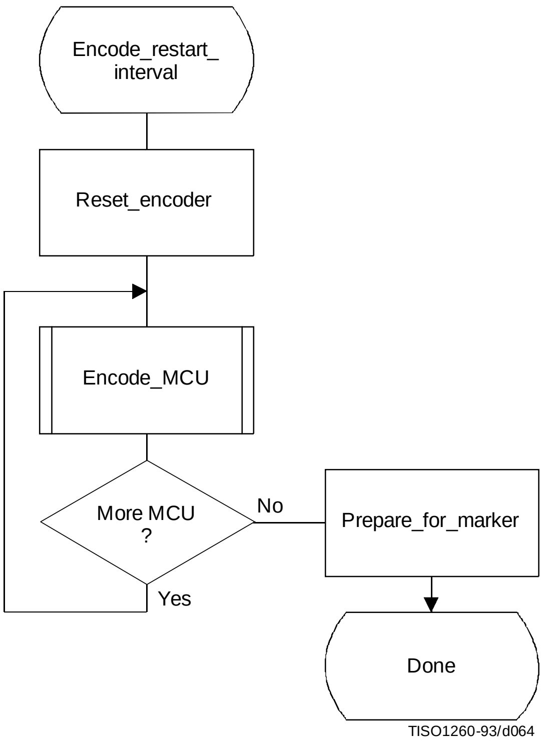


Figure E.4 – Control procedure for encoding a restart interval

The "Reset_encoder" procedure consists at least of the following:

a) if arithmetic coding is used, initialize the arithmetic encoder using the "Initenc" procedure described in D.1.7;

b) for DCT-based processes, set the DC prediction (PRED) to zero for all components in the scan (see F.1.1.5.1);

c) for lossless processes, reset the prediction to a default value for all components in the scan (see H.1.1);

d) do all other implementation-dependent setups that may be necessary.

The procedure "Prepare_for_marker" terminates the entropy-coded segment by:

a) padding a Huffman entropy-coded segment with 1-bits to complete the final byte (and if needed stuffing a zero byte) (see F.1.2.3); or

b) invoking the procedure "Flush" (see D.1.8) to terminate an arithmetic entropy-coded segment.

NOTE – The number of minimum coded units (MCU) in the final restart interval must be adjusted to match the number of MCU in the scan. The number of MCU is calculated from the frame and scan parameters. (See Annex B.)

E.1.5 Control procedure for encoding a minimum coded unit (MCU)

The minimum coded unit is defined in A.2. Within a given MCU the data units are coded in the order in which they occur in the MCU. The control procedure for encoding a MCU is shown in Figure E.5.

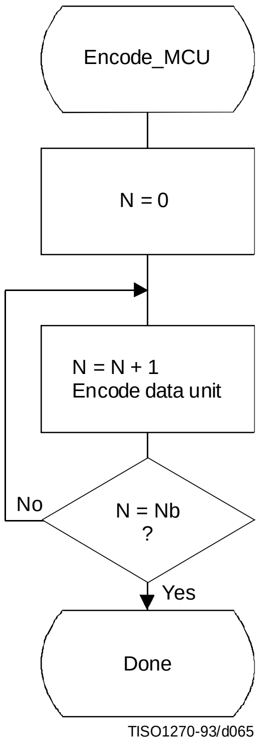


Figure E.5 – Control procedure for encoding a minimum coded unit (MCU)

In Figure E.5, Nb refers to the number of data units in the MCU. The order in which data units occur in the MCU is defined in A.2. The data unit is an 8×8 block for DCT-based processes, and a single sample for lossless processes.

The procedures for encoding a data unit are specified in Annexes F, G, and H.

E.2 Decoder control procedures

E.2.1 Control procedure for decoding compressed image data

Figure E.6 shows the decoding process control for compressed image data.

Decoding control centers around identification of various markers. The first marker must be the SOI (Start Of Image) marker. The "Decoder_setup" procedure resets the restart interval (Ri = 0) and, if the decoder has arithmetic decoding capabilities, sets the conditioning tables for the arithmetic coding to their default values. (See F.1.4.4.1.4 and F.1.4.4.2.1.) The next marker is normally a $\mathrm{SOF_n}$ (Start Of Frame) marker; if this is not found, one of the marker segments listed in Table E.1 has been received.

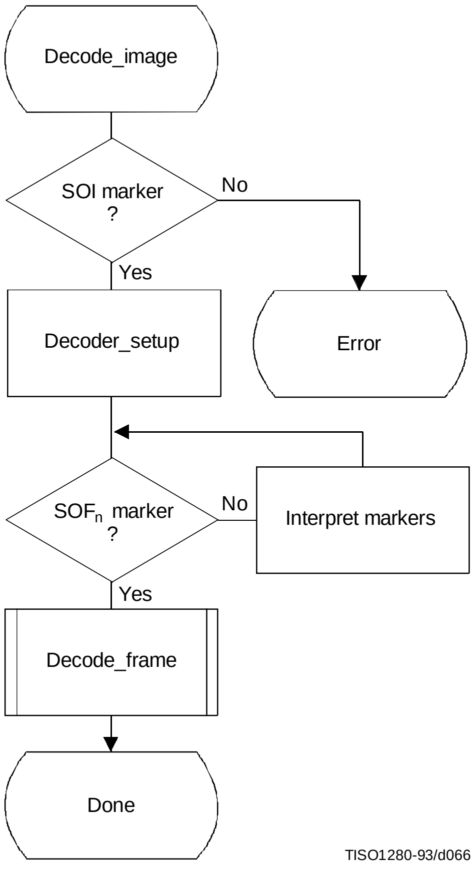


Figure E.6 – Control procedure for decoding compressed image data

Table E.1 – Markers recognized by "Interpret markers"

Marker	Purpose
DHT	Define Huffman Tables
DAC	Define Arithmetic Conditioning
DQT	Define Quantization Tables
DRI	Define Restart Interval
APP_n	Application defined marker
COM	Comment

Note that optional X'FF' fill bytes which may precede any marker shall be discarded before determining which marker is present.

The additional logic to interpret these various markers is contained in the box labeled "Interpret markers". DHT markers shall be interpreted by processes using Huffman coding. DAC markers shall be interpreted by processes using arithmetic coding. DQT markers shall be interpreted by DCT-based decoders. DRI markers shall be interpreted by all decoders. APPn and COM markers shall be interpreted only to the extent that they do not interfere with the decoding.

By definition, the procedures in "Interpret markers" leave the system at the next marker. Note that if the expected SOI marker is missing at the start of the compressed image data, an error condition has occurred. The techniques for detecting and managing error conditions can be as elaborate or as simple as desired.

E.2.2 Control procedure for decoding a frame

Figure E.7 shows the control procedure for the decoding of a frame.

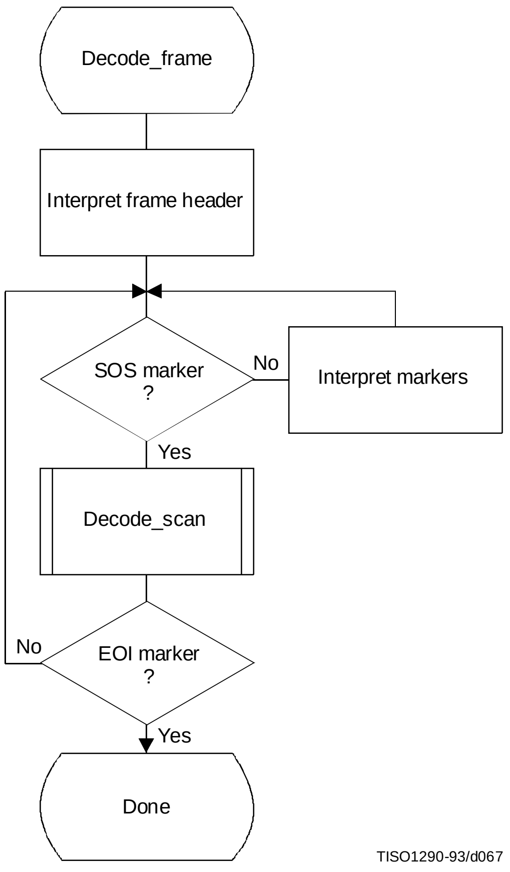


Figure E.7 – Control procedure for decoding a frame

The loop is terminated if the EOI marker is found at the end of the scan.

The markers recognized by "Interpret markers" are listed in Table E.1. Subclause E.2.1 describes the extent to which the various markers shall be interpreted.

E.2.3 Control procedure for decoding a scan

Figure E.8 shows the decoding of a scan.

The loop is terminated when the expected number of restart intervals has been decoded.

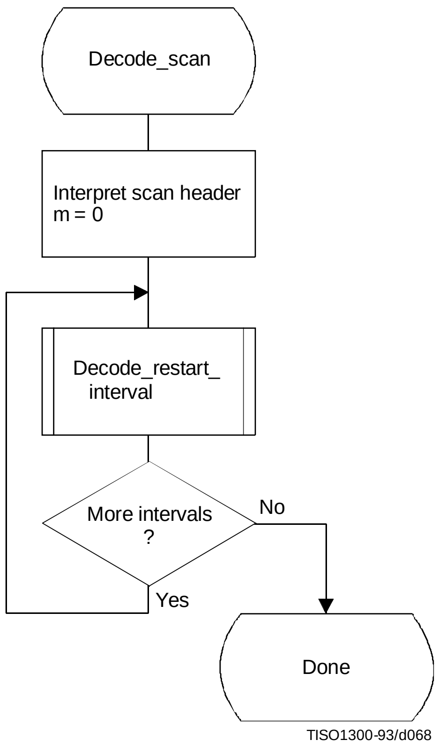


Figure E.8 – Control procedure for decoding a scan

E.2.4 Control procedure for decoding a restart interval

The procedure for decoding a restart interval is shown in Figure E.9. The “Reset_decoder” procedure consists at least of the following:

a) if arithmetic coding is used, initialize the arithmetic decoder using the “Initdec” procedure described in D.2.7;

b) for DCT-based processes, set the DC prediction (PRED) to zero for all components in the scan (see F.2.1.3.1);

c) for lossless process, reset the prediction to a default value for all components in the scan (see H.2.1);

d) do all other implementation-dependent setups that may be necessary.

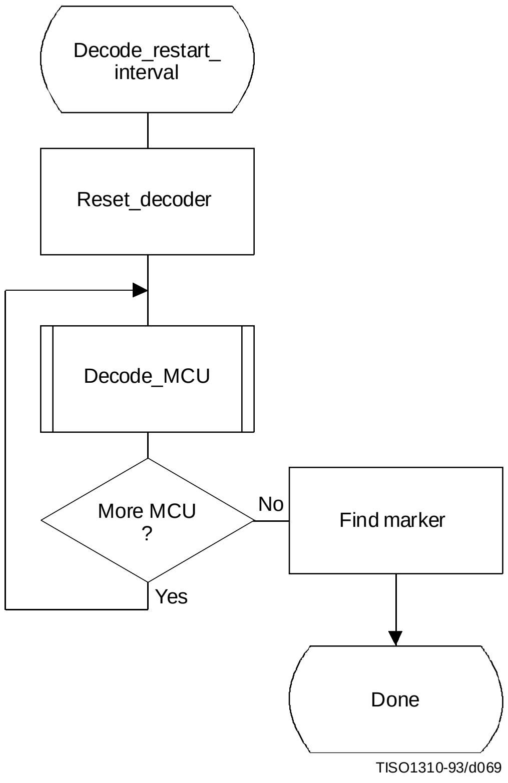


Figure E.9 – Control procedure for decoding a restart interval

At the end of the restart interval, the next marker is located. If a problem is detected in locating this marker, error handling procedures may be invoked. While such procedures are optional, the decoder shall be able to correctly recognize restart markers in the compressed data and reset the decoder when they are encountered. The decoder shall also be able to recognize the DNL marker, set the number of lines defined in the DNL segment, and end the “Decode_restart_interval” procedure.

NOTE – The final restart interval may be smaller than the size specified by the DRI marker segment, as it includes only the number of MCUs remaining in the scan.

E.2.5 Control procedure for decoding a minimum coded unit (MCU)

The procedure for decoding a minimum coded unit (MCU) is shown in Figure E.10.

In Figure E.10 Nb is the number of data units in a MCU.

The procedures for decoding a data unit are specified in Annexes F, G, and H.

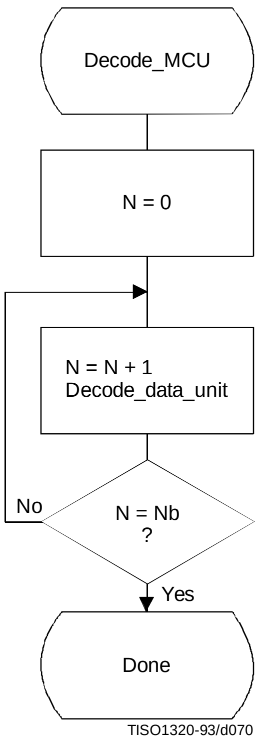


Figure E.10 – Control procedure for decoding a minimum coded unit (MCU)

Annex F

Sequential DCT-based mode of operation

(This annex forms an integral part of this Recommendation | International Standard)

This annex provides a **functional specification** of the following coding processes for the sequential DCT-based mode of operation:

1) baseline sequential;
2) extended sequential, Huffman coding, 8-bit sample precision;
3) extended sequential, arithmetic coding, 8-bit sample precision;
4) extended sequential, Huffman coding, 12-bit sample precision;
5) extended sequential, arithmetic coding, 12-bit sample precision.

For each of these, the encoding process is specified in F.1, and the decoding process is specified in F.2. The functional specification is presented by means of specific flow charts for the various procedures which comprise these coding processes.

NOTE – There is **no requirement** in this Specification that any encoder or decoder which embodies one of the above-named processes shall implement the procedures in precisely the manner specified by the flow charts in this annex. It is necessary only that an encoder or decoder implement the **function** specified in this annex. The sole criterion for an encoder or decoder to be considered in compliance with this Specification is that it satisfy the requirements given in clause 6 (for encoders) or clause 7 (for decoders), as determined by the compliance tests specified in Part 2.

F.1 Sequential DCT-based encoding processes

F.1.1 Sequential DCT-based control procedures and coding models

F.1.1.1 Control procedures for sequential DCT-based encoders

The control procedures for encoding an image and its constituent parts – the frame, scan, restart interval and MCU – are given in Figures E.1 to E.5. The procedure for encoding a MCU (see Figure E.5) repetitively calls the procedure for encoding a data unit. For DCT-based encoders the data unit is an 8×8 block of samples.

F.1.1.2 Procedure for encoding an 8×8 block data unit

For the sequential DCT-based processes encoding an 8×8 block data unit consists of the following procedures:

a) level shift, calculate forward 8×8 DCT and quantize the resulting coefficients using table destination specified in frame header;
b) encode DC coefficient for 8×8 block using DC table destination specified in scan header;
c) encode AC coefficients for 8×8 block using AC table destination specified in scan header.

F.1.1.3 Level shift and forward DCT (FDCT)

The mathematical definition of the FDCT is given in A.3.3.

Prior to computing the FDCT the input data are level shifted to a signed two’s complement representation as described in A.3.1. For 8-bit input precision the level shift is achieved by subtracting 128. For 12-bit input precision the level shift is achieved by subtracting 2048.

F.1.1.4 Quantization of the FDCT

The uniform quantization procedure described in Annex A is used to quantize the DCT coefficients. One of four quantization tables may be used by the encoder. No default quantization tables are specified in this Specification. However, some typical quantization tables are given in Annex K.

The quantized DCT coefficient values are signed, two’s complement integers with 11-bit precision for 8-bit input precision and 15-bit precision for 12-bit input precision.

F.1.1.5 Encoding models for the sequential DCT procedures

The two dimensional array of quantized DCT coefficients is rearranged in a zig-zag sequence order defined in A.3.6. The zig-zag order coefficients are denoted ZZ (0) through ZZ(63) with:

$$ZZ(0) = Sq_{00}, ZZ(1) = Sq_{01}, ZZ(2) = Sq_{10}, \bullet, \bullet, \bullet, ZZ(63) = Sq_{77}$$

Sq_{vu} are defined in Figure A.6.

Two coding procedures are used, one for the DC coefficient ZZ(0) and the other for the AC coefficients ZZ(1)..ZZ(63). The coefficients are encoded in the order in which they occur in zig-zag sequence order, starting with the DC coefficient. The coefficients are represented as two's complement integers.

F.1.1.5.1 Encoding model for DC coefficients

The DC coefficients are coded differentially, using a one-dimensional predictor, PRED, which is the quantized DC value from the most recently coded 8×8 block from the same component. The difference, DIFF, is obtained from

$$DIFF = ZZ(0) – PRED$$

At the beginning of the scan and at the beginning of each restart interval, the prediction for the DC coefficient prediction is initialized to 0. (Recall that the input data have been level shifted to two's complement representation.)

F.1.1.5.2 Encoding model for AC coefficients

Since many coefficients are zero, runs of zeros are identified and coded efficiently. In addition, if the remaining coefficients in the zig-zag sequence order are all zero, this is coded explicitly as an end-of-block (EOB).

F.1.2 Baseline Huffman encoding procedures

The baseline encoding procedure is for 8-bit sample precision. The encoder may employ up to two DC and two AC Huffman tables within one scan.

F.1.2.1 Huffman encoding of DC coefficients

F.1.2.1.1 Structure of DC code table

The DC code table consists of a set of Huffman codes (maximum length 16 bits) and appended additional bits (in most cases) which can code any possible value of DIFF, the difference between the current DC coefficient and the prediction. The Huffman codes for the difference categories are generated in such a way that no code consists entirely of 1-bits (X'FF' prefix marker code avoided).

The two's complement difference magnitudes are grouped into 12 categories, SSSS, and a Huffman code is created for each of the 12 difference magnitude categories (see Table F.1).

For each category, except SSSS = 0, an additional bits field is appended to the code word to uniquely identify which difference in that category actually occurred. The number of extra bits is given by SSSS; the extra bits are appended to the LSB of the preceding Huffman code, most significant bit first. When DIFF is positive, the SSSS low order bits of DIFF are appended. When DIFF is negative, the SSSS low order bits of (DIFF – 1) are appended. Note that the most significant bit of the appended bit sequence is 0 for negative differences and 1 for positive differences.

F.1.2.1.2 Defining Huffman tables for the DC coefficients

The syntax for specifying the Huffman tables is given in Annex B. The procedure for creating a code table from this information is described in Annex C. No more than two Huffman tables may be defined for coding of DC coefficients. Two examples of Huffman tables for coding of DC coefficients are provided in Annex K.

Table F.1 – Difference magnitude categories for DC coding

SSSS	DIFF values
0	0
1	–1,1
2	–3,–2,2,3
3	–7..–4,4..7
4	–15..–8,8..15
5	–31..–16,16..31
6	–63..–32,32..63
7	–127..–64,64..127
8	–255..–128,128..255
9	–511..–256,256..511
10	–1 023..–512,512..1 023
11	–2 047..–1 024,1 024..2 047

F.1.2.1.3 Huffman encoding procedures for DC coefficients

The encoding procedure is defined in terms of a set of extended tables, XHUFCO and XHUFSI, which contain the complete set of Huffman codes and sizes for all possible difference values. For full 12-bit precision the tables are relatively large. For the baseline system, however, the precision of the differences may be small enough to make this description practical.

XHUFCO and XHUFSI are generated from the encoder tables EHUFCO and EHUFSI (see Annex C) by appending to the Huffman codes for each difference category the additional bits that completely define the difference. By definition, XHUFCO and XHUFSI have entries for each possible difference value. XHUFCO contains the concatenated bit pattern of the Huffman code and the additional bits field; XHUFSI contains the total length in bits of this concatenated bit pattern. Both are indexed by DIFF, the difference between the DC coefficient and the prediction.

The Huffman encoding procedure for the DC difference, DIFF, is:

SIZE = XHUFSI(DIFF)

CODE = XHUFCO(DIFF)

code SIZE bits of CODE

where DC is the quantized DC coefficient value and PRED is the predicted quantized DC value. The Huffman code (CODE) (including any additional bits) is obtained from XHUFCO and SIZE (length of the code including additional bits) is obtained from XHUFSI, using DIFF as the index to the two tables.

F.1.2.2 Huffman encoding of AC coefficients

F.1.2.2.1 Structure of AC code table

Each non-zero AC coefficient in ZZ is described by a composite 8-bit value, RS, of the form

RS = binary ’RRRRSSSS’

The 4 least significant bits, 'SSSS', define a category for the amplitude of the next non-zero coefficient in ZZ, and the 4 most significant bits, 'RRRR', give the position of the coefficient in ZZ relative to the previous non-zero coefficient (i.e. the run-length of zero coefficients between non-zero coefficients). Since the run length of zero coefficients may exceed 15, the value 'RRRRSSSS' = X'F0' is defined to represent a run length of 15 zero coefficients followed by a coefficient of zero amplitude. (This can be interpreted as a run length of 16 zero coefficients.) In addition, a special value 'RRRRSSSS' = '00000000' is used to code the end-of-block (EOB), when all remaining coefficients in the block are zero.

The general structure of the code table is illustrated in Figure F.1. The entries marked "N/A" are undefined for the baseline procedure.

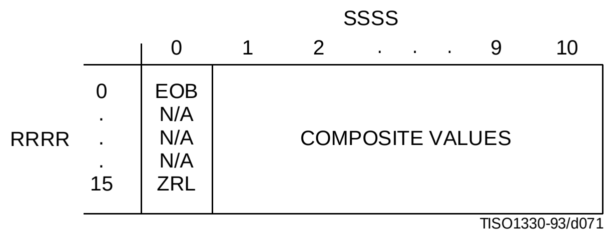


Figure F.1 – Two-dimensional value array for Huffman coding

The magnitude ranges assigned to each value of SSSS are defined in Table F.2.

Table F.2 – Categories assigned to coefficient values

SSSS	AC coefficients
1	–1,1
2	–3,–2,2,3
3	–7..–4,4..7
4	–15..–8,8..15
5	–31..–16,16..31
6	–63..–32,32..63
7	–127..–64,64..127
8	–255..–128,128..255
9	–511..–256,256..511
10	–1 023..–512,512..1 023

The composite value, RRRRSSSS, is Huffman coded and each Huffman code is followed by additional bits which specify the sign and exact amplitude of the coefficient.

The AC code table consists of one Huffman code (maximum length 16 bits, not including additional bits) for each possible composite value. The Huffman codes for the 8-bit composite values are generated in such a way that no code consists entirely of 1-bits.

The format for the additional bits is the same as in the coding of the DC coefficients. The value of SSSS gives the number of additional bits required to specify the sign and precise amplitude of the coefficient. The additional bits are either the low-order SSSS bits of ZZ(K) when ZZ(K) is positive or the low-order SSSS bits of ZZ(K) – 1 when ZZ(K) is negative. ZZ(K) is the *K*th coefficient in the zig-zag sequence of coefficients being coded.

F.1.2.2.2 Defining Huffman tables for the AC coefficients

The syntax for specifying the Huffman tables is given in Annex B. The procedure for creating a code table from this information is described in Annex C.

In the baseline system no more than two Huffman tables may be defined for coding of AC coefficients. Two examples of Huffman tables for coding of AC coefficients are provided in Annex K.

F.1.2.2.3 Huffman encoding procedures for AC coefficients

As defined in Annex C, the Huffman code table is assumed to be available as a pair of tables, EHUFCO (containing the code bits) and EHUFSI (containing the length of each code in bits), both indexed by the composite value defined above.

The procedure for encoding the AC coefficients in a block is shown in Figures F.2 and F.3. In Figure F.2, K is the index to the zig-zag scan position and R is the run length of zero coefficients.

The procedure “Append EHUFSI(X’F0’) bits of EHUFCO(X’F0’)” codes a run of 16 zero coefficients (ZRL code of Figure F.1). The procedure “Code EHUFSI(0) bits of EHUFCO(0)” codes the end-of-block (EOB code). If the last coefficient (K = 63) is not zero, the EOB code is bypassed.

CSIZE is a procedure which maps an AC coefficient to the SSSS value as defined in Table F.2.

F.1.2.3 Byte stuffing

In order to provide code space for marker codes which can be located in the compressed image data without decoding, byte stuffing is used.

Whenever, in the course of normal encoding, the byte value X’FF’ is created in the code string, a X’00’ byte is stuffed into the code string.

If a X’00’ byte is detected after a X’FF’ byte, the decoder must discard it. If the byte is not zero, a marker has been detected, and shall be interpreted to the extent needed to complete the decoding of the scan.

Byte alignment of markers is achieved by padding incomplete bytes with 1-bits. If padding with 1-bits creates a X’FF’ value, a zero byte is stuffed before adding the marker.

F.1.3 Extended sequential DCT-based Huffman encoding process for 8-bit sample precision

This process is identical to the Baseline encoding process described in F.1.2, with the exception that the number of sets of Huffman table destinations which may be used within the same scan is increased to four. Four DC and four AC Huffman table destinations is the maximum allowed by this Specification.

F.1.4 Extended sequential DCT-based arithmetic encoding process for 8-bit sample precision

This subclause describes the use of arithmetic coding procedures in the sequential DCT-based encoding process.

NOTE – The arithmetic coding procedures in this Specification are defined for the maximum precision to encourage interchangeability.

The arithmetic coding extensions have the same DCT model as the Baseline DCT encoder. Therefore, Annex F.1.1 also applies to arithmetic coding. As with the Huffman coding technique, the binary arithmetic coding technique is lossless. It is possible to transcode between the two systems without either FDCT or IDCT computations, and without modification of the reconstructed image.

The basic principles of adaptive binary arithmetic coding are described in Annex D. Up to four DC and four AC conditioning table destinations and associated statistics areas may be used within one scan.

The arithmetic encoding procedures for encoding binary decisions, initializing the statistics area, initializing the encoder, terminating the code string, and adding restart markers are listed in Table D.1 of Annex D.

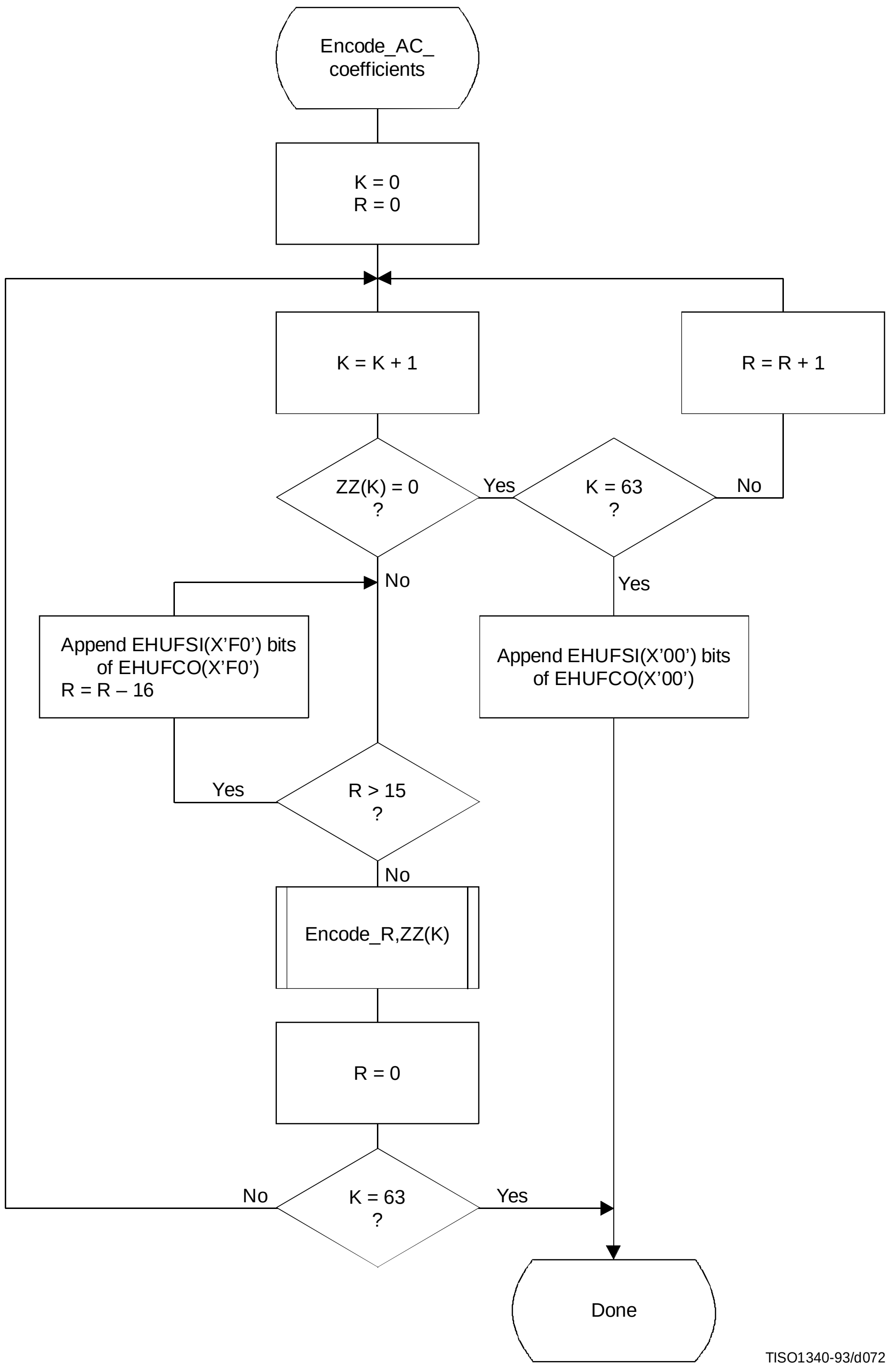


Figure F.2 – Procedure for sequential encoding of AC coefficients with Huffman coding

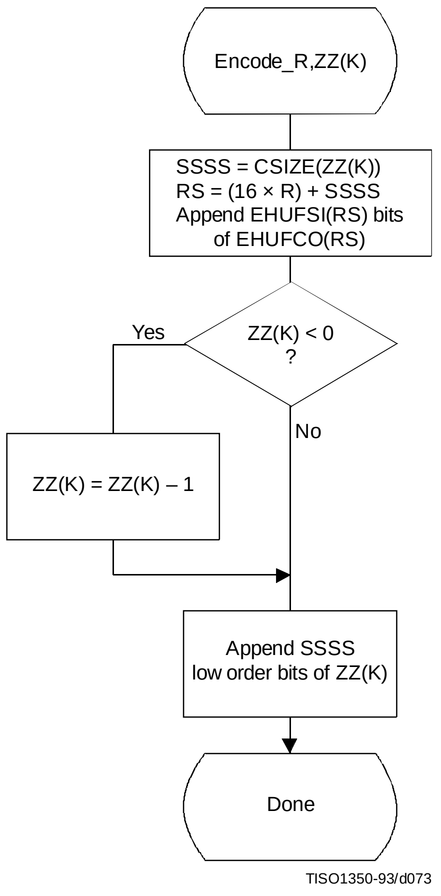


Figure F.3 – Sequential encoding of a non-zero AC coefficient

Some of the procedures in Table D.1 are used in the higher level control structure for scans and restart intervals described in Annex E. At the beginning of scans and restart intervals, the probability estimates used in the arithmetic coder are reset to the standard initial value as part of the Initenc procedure which restarts the arithmetic coder. At the end of scans and restart intervals, the Flush procedure is invoked to empty the code register before the next marker is appended.

F.1.4.1 Arithmetic encoding of DC coefficients

The basic structure of the decision sequence for encoding a DC difference value, DIFF, is shown in Figure F.4.

The context-index S0 and other context-indices used in the DC coding procedures are defined in Table F.4 (see F.1.4.4.1.3). A 0-decision is coded if the difference value is zero and a 1-decision is coded if the difference is not zero. If the difference is not zero, the sign and magnitude are coded using the procedure Encode_V(S0), which is described in F.1.4.3.1.

F.1.4.2 Arithmetic encoding of AC coefficients

The AC coefficients are coded in the order in which they occur in the zig-zag sequence ZZ(1,...,63). An end-of-block (EOB) binary decision is coded before coding the first AC coefficient in ZZ, and after each non-zero coefficient. If the EOB occurs, all remaining coefficients in ZZ are zero. Figure F.5 illustrates the decision sequence. The equivalent procedure for the Huffman coder is found in Figure F.2.

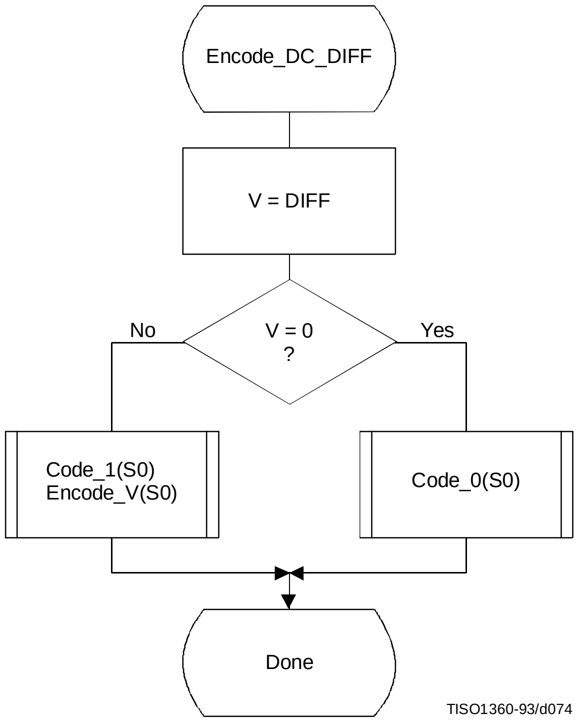


Figure F.4 – Coding model for arithmetic coding of DC difference

The context-indices SE and S0 used in the AC coding procedures are defined in Table F.5 (see F.1.4.4.2). In Figure F.5, K is the index to the zig-zag sequence position. For the sequential scan, Kmin is 1 and Se is 63. The V = 0 decision is part of a loop which codes runs of zero coefficients. Whenever the coefficient is non-zero, "Encode_V(S0)" codes the sign and magnitude of the coefficient. Each time a non-zero coefficient is coded, it is followed by an EOB decision. If the EOB occurs, a 1-decision is coded to indicate that the coding of the block is complete. If the coefficient for K = Se is not zero, the EOB decision is skipped.

F.1.4.3 Encoding the binary decision sequence for non-zero DC differences and AC coefficients

Both the DC difference and the AC coefficients are represented as signed two's complement integer values. The decomposition of these signed integer values into a binary decision tree is done in the same way for both the DC and AC coding models.

Although the binary decision trees for this section of the DC and AC coding models are the same, the statistical models for assigning statistics bins to the binary decisions in the tree are quite different.

F.1.4.3.1 Structure of the encoding decision sequence

The encoding sequence can be separated into three procedures, a procedure which encodes the sign, a second procedure which identifies the magnitude category, and a third procedure which identifies precisely which magnitude occurred within the category identified in the second procedure.

At the point where the binary decision sequence in Encode_V(S0) starts, the coefficient or difference has already been determined to be non-zero. That determination was made in the procedures in Figures F.4 and F.5.

Denoting either DC differences (DIFF) or AC coefficients as V, the non-zero signed integer value of V is encoded by the sequence shown in Figure F.6. This sequence first codes the sign of V. It then (after converting V to a magnitude and decrementing it by 1 to give Sz) codes the magnitude category of Sz (code_log2_Sz), and then codes the low order magnitude bits (code_Sz_bits) to identify the exact magnitude value.

There are two significant differences between this sequence and the similar set of operations described in F.1.2 for Huffman coding. First, the sign is encoded before the magnitude category is identified, and second, the magnitude is decremented by 1 before the magnitude category is identified.

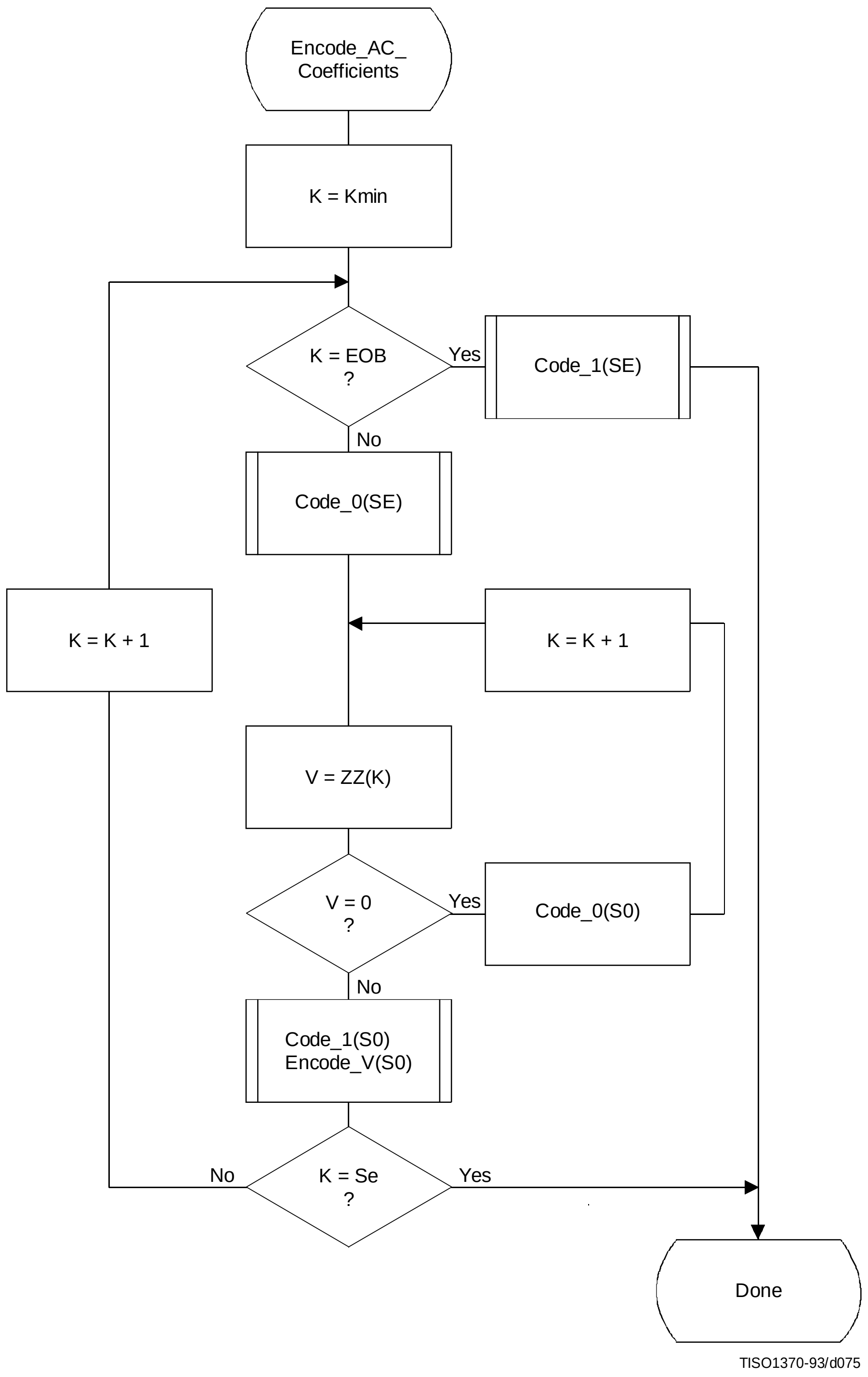


Figure F.5 – AC coding model for arithmetic coding

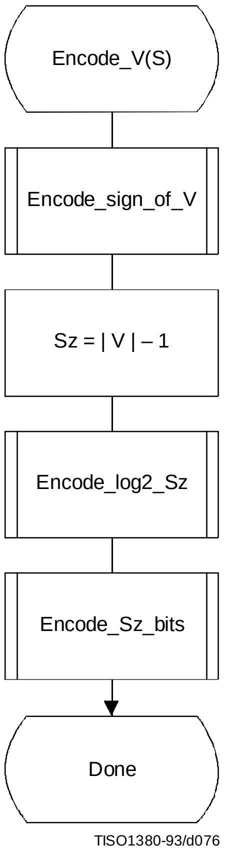


Figure F.6 – Sequence of procedures in encoding non-zero values of V

F.1.4.3.1.1 Encoding the sign

The sign is encoded by coding a 0-decision when the sign is positive and a 1-decision when the sign is negative (see Figure F.7).

The context-indices SS, SN and SP are defined for DC coding in Table F.4 and for AC coding in Table F.5. After the sign is coded, the context-index S is set to either SN or SP, establishing an initial value for Encode_log2_Sz.

F.1.4.3.1.2 Encoding the magnitude category

The magnitude category is determined by a sequence of binary decisions which compares Sz against an exponentially increasing bound (which is a power of 2) in order to determine the position of the leading 1-bit. This establishes the magnitude category in much the same way that the Huffman encoder generates a code for the value associated with the difference category. The flow chart for this procedure is shown in Figure F.8.

The starting value of the context-index S is determined in Encode_sign_of_V, and the context-index values X1 and X2 are defined for DC coding in Table F.4 and for AC coding in Table F.5. In Figure F.8, M is the exclusive upper bound for the magnitude and the abbreviations “SLL” and “SRL” refer to the shift-left-logical and shift-right-logical operations – in this case by one bit position. The SRL operation at the completion of the procedure aligns M with the most significant bit of Sz (see Table F.3).

The highest precision allowed for the DCT is 15 bits. Therefore, the highest precision required for the coding decision tree is 16 bits for the DC coefficient difference and 15 bits for the AC coefficients, including the sign bit.

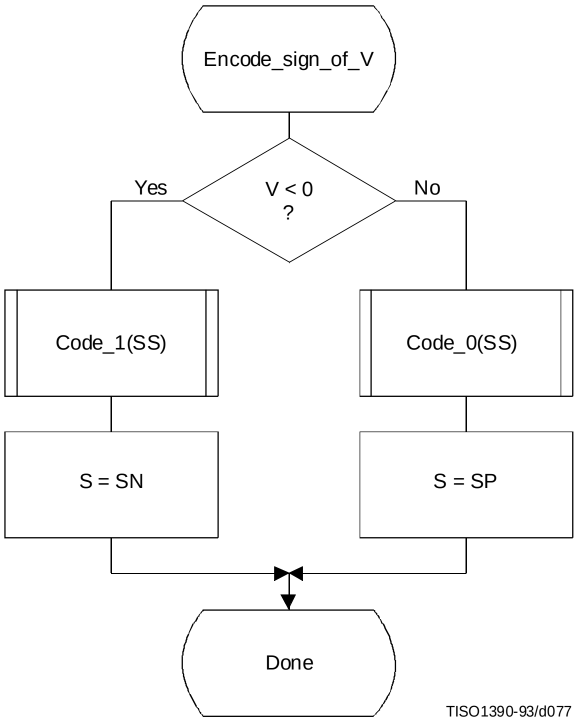


Figure F.7 – Encoding the sign of V

Table F.3 – Categories for each maximum bound

Exclusive upper bound (M)	Sz range	Number of low order magnitude bits
1	0	0
2	1	0
4	2,3	1
8	4,...,7	2
16	8,...,15	3
32	16,...,31	4
64	32,...,63	5
128	64,...,127	6
256	128,...,255	7
512	256,...,511	8
1 024	512,...,1 023	9
2 048	1 024,...,2 047	10
4 096	2 048,...,4 095	11
8 192	4 096,...,8 191	12
16 384	8 192,...,16 383	13
32 768	16 384,...,32 767	14

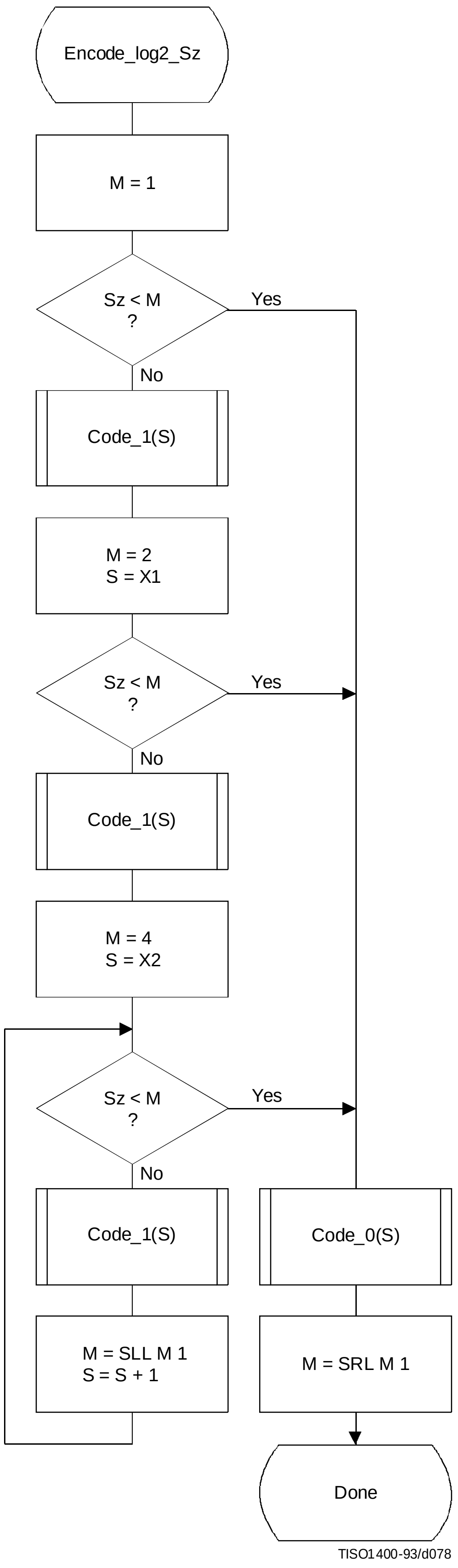


Figure F.8 – Decision sequence to establish the magnitude category

F.1.4.3.1.3 Encoding the exact value of the magnitude

After the magnitude category is encoded, the low order magnitude bits are encoded. These bits are encoded in order of decreasing bit significance. The procedure is shown in Figure F.9. The abbreviation “SRL” indicates the shift-right-logical operation, and M is the exclusive bound established in Figure F.8. Note that M has only one bit set – shifting M right converts it into a bit mask for the logical “AND” operation.

The starting value of the context-index S is determined in Encode_log2_Sz. The increment of S by 14 at the beginning of this procedure sets the context-index to the value required in Tables F.4 and F.5.

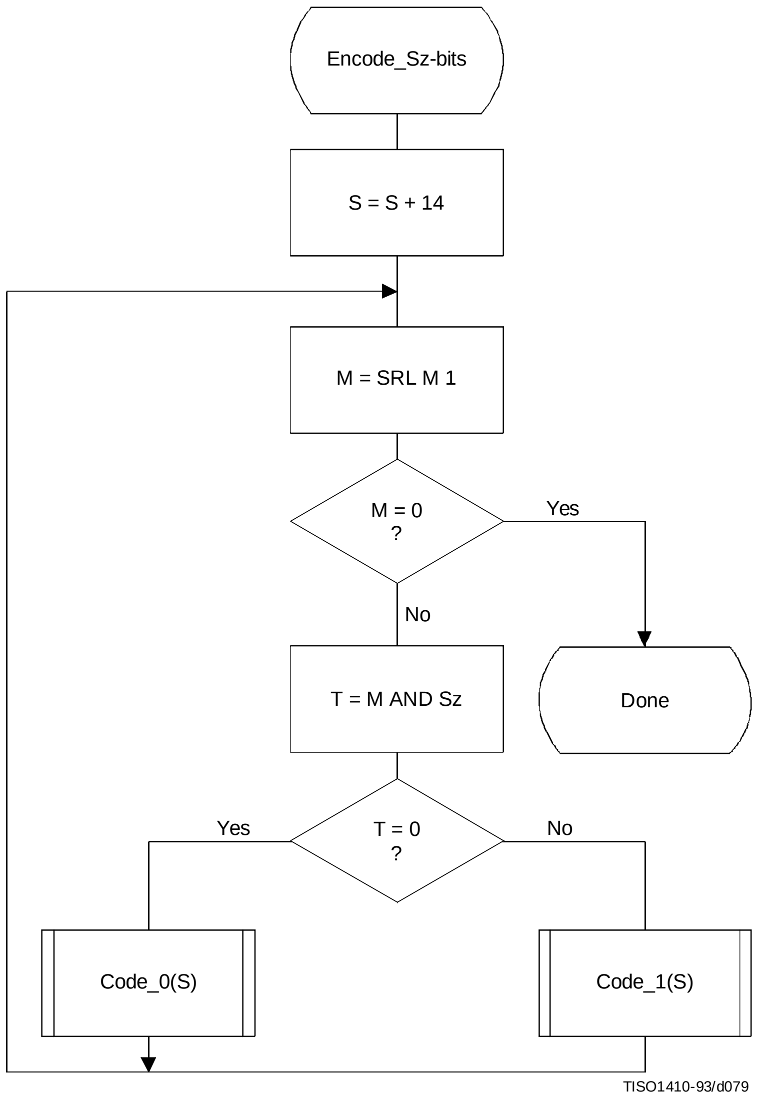


Figure F.9 – Decision sequence to code the magnitude bit pattern

F.1.4.4 Statistical models

An adaptive binary arithmetic coder requires a statistical model. The statistical model defines the contexts which are used to select the conditional probability estimates used in the encoding and decoding procedures.

Each decision in the binary decision trees is associated with one or more contexts. These contexts identify the sense of the MPS and the index in Table D.3 of the conditional probability estimate Qe which is used to encode and decode the binary decision.

The arithmetic coder is adaptive, which means that the probability estimates for each context are developed and maintained by the arithmetic coding system on the basis of prior coding decisions for that context.

F.1.4.4.1 Statistical model for coding DC prediction differences

The statistical model for coding the DC difference conditions some of the probability estimates for the binary decisions on previous DC coding decisions.

F.1.4.4.1.1 Statistical conditioning on sign

In coding the DC coefficients, four separate statistics bins (probability estimates) are used in coding the zero/not-zero ($V = 0$) decision, the sign decision and the first magnitude category decision. Two of these bins are used to code the $V = 0$ decision and the sign decision. The other two bins are used in coding the first magnitude decision, $Sz < 1$; one of these bins is used when the sign is positive, and the other is used when the sign is negative. Thus, the first magnitude decision probability estimate is conditioned on the sign of V.

F.1.4.4.1.2 Statistical conditioning on DC difference in previous block

The probability estimates for these first three decisions are also conditioned on Da, the difference value coded for the previous DCT block of the same component. The differences are classified into five groups: zero, small positive, small negative, large positive and large negative. The relationship between the default classification and the quantization scale is shown in Figure F.10.

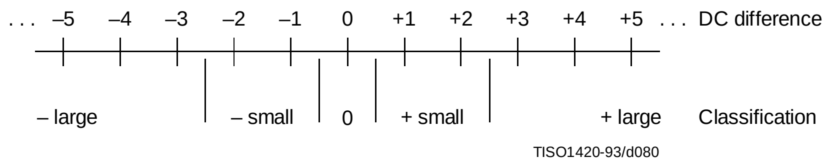


Figure F.10 – Conditioning classification of difference values

The bounds for the "small" difference category determine the classification. Defining L and U as integers in the range 0 to 15 inclusive, the lower bound (exclusive) for difference magnitudes classified as "small" is zero for $L = 0$, and is 2^{L-1} for $L > 0$.

The upper bound (inclusive) for difference magnitudes classified as "small" is 2^U.

L shall be less than or equal to U.

These bounds for the conditioning category provide a segmentation which is identical to that listed in Table F.3.

F.1.4.4.1.3 Assignment of statistical bins to the DC binary decision tree

As shown in Table F.4, each statistics area for DC coding consists of a set of 49 statistics bins. In the following explanation, it is assumed that the bins are contiguous. The first 20 bins consist of five sets of four bins selected by a context-index S0. The value of S0 is given by DC_Context(Da), which provides a value of 0, 4, 8, 12 or 16, depending on the difference classification of Da (see F.1.4.4.1.2). The remaining 29 bins, X1,...,X15,M2,...,M15, are used to code magnitude category decisions and magnitude bits.

Table F.4 – Statistical model for DC coefficient coding

Context-index	Value	Coding decision
S0	DC_Context(Da)	V = 0
SS	S0 + 1	Sign of V
SP	S0 + 2	Sz < 1 if V > 0
SN	S0 + 3	Sz < 1 if V < 0
X1	20	Sz < 2
X2	X1 + 1	Sz < 4
X3	X1 + 2	Sz < 8
.	.	.
.	.	.
X15	X1 + 14	Sz < 2^{15}
M2	X2 + 14	Magnitude bits if Sz < 4
M3	X3 + 14	Magnitude bits if Sz < 8
.	.	.
.	.	.
M15	X15 + 14	Magnitude bits if Sz < 2^{15}

F.1.4.4.1.4 Default conditioning for DC statistical model

The bounds, L and U, for determining the conditioning category have the default values L = 0 and U = 1. Other bounds may be set using the DAC (Define Arithmetic coding Conditioning) marker segment, as described in Annex B.

F.1.4.4.1.5 Initial conditions for DC statistical model

At the start of a scan and at the beginning of each restart interval, the difference for the previous DC value is defined to be zero in determining the conditioning state.

F.1.4.4.2 Statistical model for coding the AC coefficients

As shown in Table F.5, each statistics area for AC coding consists of a contiguous set of 245 statistics bins. Three bins are used for each value of the zig-zag index K, and two sets of 28 additional bins X2,...,X15,M2,...,M15 are used for coding the magnitude category and magnitude bits.

The value of SE (and also S0, SP and SN) is determined by the zig-zag index K. Since K is in the range 1 to 63, the lowest value for SE is 0 and the largest value for SP is 188. SS is not assigned a value in AC coefficient coding, as the signs of the coefficients are coded with a fixed probability value of approximately 0.5 (Qe = X'5A1D', MPS = 0).

The value of X2 is given by AC_Context(K). This gives X2 = 189 when K ≤ Kx and X2 = 217 when K > Kx, where Kx is defined using the DAC marker segment (see B.2.4.3).

Note that a X1 statistics bin is not used in this sequence. Instead, the 63 × 1 array of statistics bins for the magnitude category is used for two decisions. Once the magnitude bound has been determined – at statistics bin Xn, for example – a single statistics bin, Mn, is used to code the magnitude bit sequence for that bound.

F.1.4.4.2.1 Default conditioning for AC coefficient coding

The default value of Kx is 5. This may be modified using the DAC marker segment, as described in Annex B.

F.1.4.4.2.2 Initial conditions for AC statistical model

At the start of a scan and at each restart, all statistics bins are re-initialized to the standard default value described in Annex D.

Table F.5 – Statistical model for AC coefficient coding

Context-index	Value	Coding decision
SE	$3 \times (K - 1)$	K = EOB
S0	SE + 1	V = 0
SS	Fixed estimate	Sign of V
SN,SP	S0 + 1	$Sz < 1$
X1	S0 + 1	$Sz < 2$
X2	AC_Context(K)	$Sz < 4$
X3	X2 + 1	$Sz < 8$
.	.	.
.	.	.
X15	X2 + 13	$Sz < 2^{15}$
M2	X2 + 14	Magnitude bits if $Sz < 4$
M3	X3 + 14	Magnitude bits if $Sz < 8$
.	.	.
.	.	.
M15	X15 + 14	Magnitude bits if $Sz < 2^{15}$

F.1.5 Extended sequential DCT-based Huffman encoding process for 12-bit sample precision

This process is identical to the sequential DCT process for 8-bit precision extended to four Huffman table destinations as documented in F.1.3, with the following changes.

F.1.5.1 Structure of DC code table for 12-bit sample precision

The two’s complement difference magnitudes are grouped into 16 categories, SSSS, and a Huffman code is created for each of the 16 difference magnitude categories.

The Huffman table for DC coding (see Table F.1) is extended as shown in Table F.6.

Table F.6 – Difference magnitude categories for DC coding

SSSS	Difference values
12	–4 095..–2 048,2 048..4 095
13	–8 191..–4 096,4 096..8 191
14	–16 383..–8 192,8 192..16 383
15	–32 767..–16 384,16 384..32 767

F.1.5.2 Structure of AC code table for 12-bit sample precision

The general structure of the code table is extended as illustrated in Figure F.11. The Huffman table for AC coding is extended as shown in Table F.7.

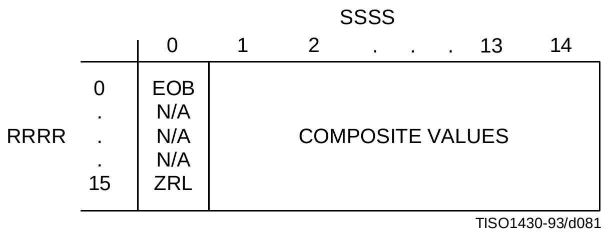


Figure F.11 – Two-dimensional value array for Huffman coding

Table F.7 – Values assigned to coefficient amplitude ranges

SSSS	AC coefficients
11	–2 047..–1 024,1 024..2 047
12	–4 095..–2 048,2 048..4 095
13	–8 191..–4 096,4 096..8 191
14	–16 383..–8 192,8 192..16 383

F.1.6 Extended sequential DCT-based arithmetic encoding process for 12-bit sample precision

The process is identical to the sequential DCT process for 8-bit precision except for changes in the precision of the FDCT computation.

The structure of the encoding procedure is identical to that specified in F.1.4 which was already defined for a 12-bit sample precision.

F.2 Sequential DCT-based decoding processes

F.2.1 Sequential DCT-based control procedures and coding models

F.2.1.1 Control procedures for sequential DCT-based decoders

The control procedures for decoding compressed image data and its constituent parts – the frame, scan, restart interval and MCU – are given in Figures E.6 to E.10. The procedure for decoding a MCU (Figure E.10) repetitively calls the procedure for decoding a data unit. For DCT-based decoders the data unit is an 8×8 block of samples.

F.2.1.2 Procedure for decoding an 8×8 block data unit

In the sequential DCT-based decoding process, decoding an 8×8 block data unit consists of the following procedures:

a) decode DC coefficient for 8×8 block using the DC table destination specified in the scan header;

b) decode AC coefficients for 8×8 block using the AC table destination specified in the scan header;

c) dequantize using table destination specified in the frame header and calculate the inverse 8×8 DCT.

F.2.1.3 Decoding models for the sequential DCT procedures

Two decoding procedures are used, one for the DC coefficient ZZ(0) and the other for the AC coefficients ZZ(1)...ZZ(63). The coefficients are decoded in the order in which they occur in the zig-zag sequence order, starting with the DC coefficient. The coefficients are represented as two's complement integers.

F.2.1.3.1 Decoding model for DC coefficients

The decoded difference, DIFF, is added to PRED, the DC value from the most recently decoded 8×8 block from the same component. Thus ZZ(0) = PRED + DIFF.

At the beginning of the scan and at the beginning of each restart interval, the prediction for the DC coefficient is initialized to zero.

F.2.1.3.2 Decoding model for AC coefficients

The AC coefficients are decoded in the order in which they occur in ZZ. When the EOB is decoded, all remaining coefficients in ZZ are initialized to zero.

F.2.1.4 Dequantization of the quantized DCT coefficients

The dequantization of the quantized DCT coefficients as described in Annex A, is accomplished by multiplying each quantized coefficient value by the quantization table value for that coefficient. The decoder shall be able to use up to four quantization table destinations.

F.2.1.5 Inverse DCT (IDCT)

The mathematical definition of the IDCT is given in A.3.3.

After computation of the IDCT, the signed output samples are level-shifted, as described in Annex A, converting the output to an unsigned representation. For 8-bit precision the level shift is performed by adding 128. For 12-bit precision the level shift is performed by adding 2 048. If necessary, the output samples shall be clamped to stay within the range appropriate for the precision (0 to 255 for 8-bit precision and 0 to 4 095 for 12-bit precision).

F.2.2 Baseline Huffman Decoding procedures

The baseline decoding procedure is for 8-bit sample precision. The decoder shall be capable of using up to two DC and two AC Huffman tables within one scan.

F.2.2.1 Huffman decoding of DC coefficients

The decoding procedure for the DC difference, DIFF, is:

```
T = DECODE

DIFF = RECEIVE(T)

DIFF = EXTEND(DIFF,T)
```

where DECODE is a procedure which returns the 8-bit value associated with the next Huffman code in the compressed image data (see F.2.2.3) and RECEIVE(T) is a procedure which places the next T bits of the serial bit string into the low order bits of DIFF, MSB first. If T is zero, DIFF is set to zero. EXTEND is a procedure which converts the partially decoded DIFF value of precision T to the full precision difference. EXTEND is shown in Figure F.12.

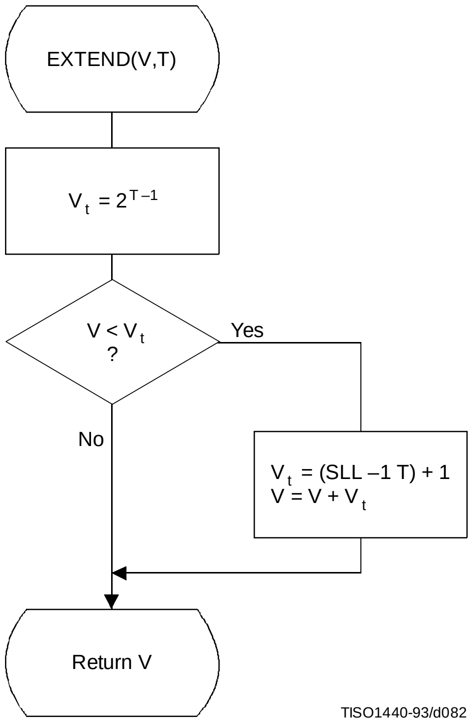


Figure F.12 – Extending the sign bit of a decoded value in V

F.2.2.2 Decoding procedure for AC coefficients

The decoding procedure for AC coefficients is shown in Figures F.13 and F.14.

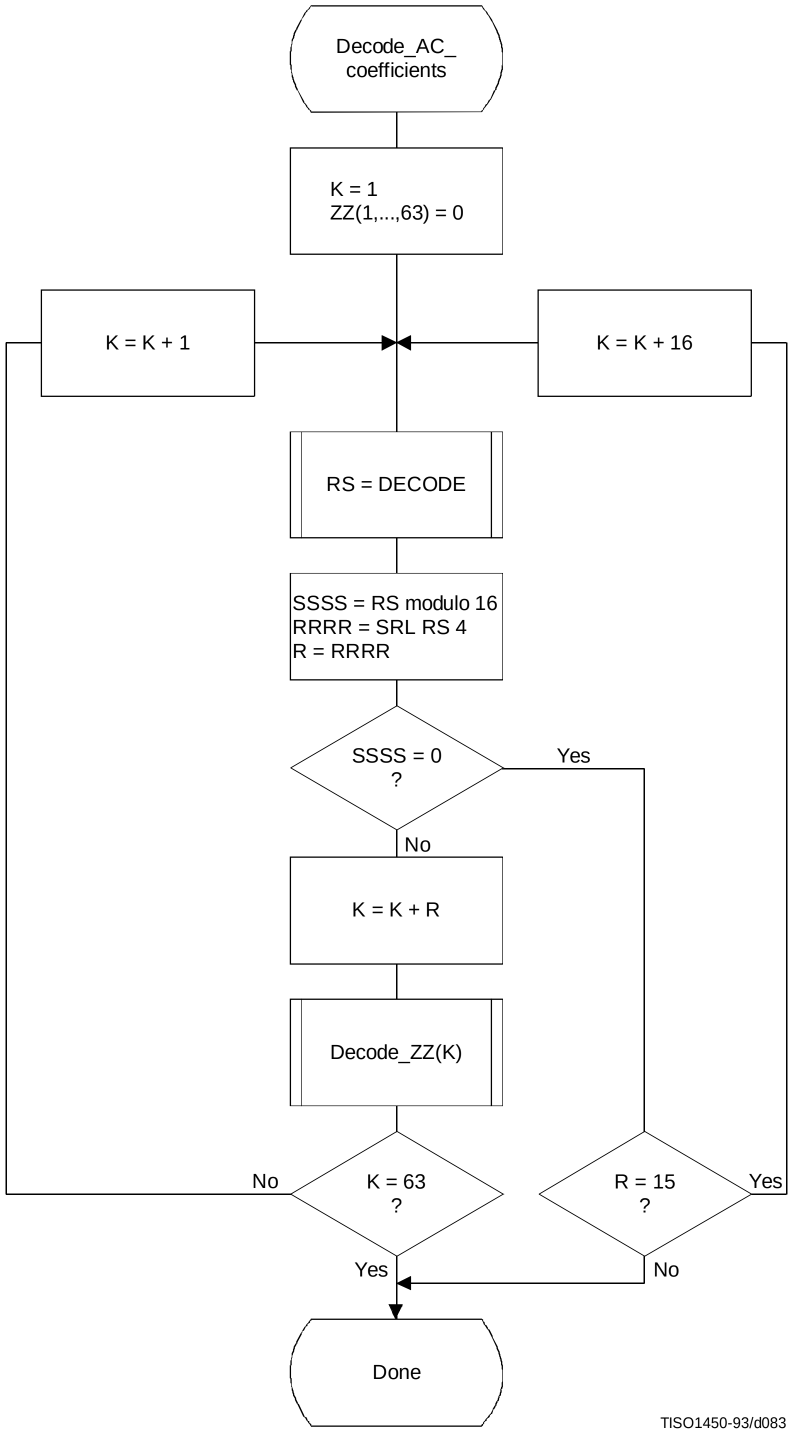


Figure F.13 – Huffman decoding procedure for AC coefficients

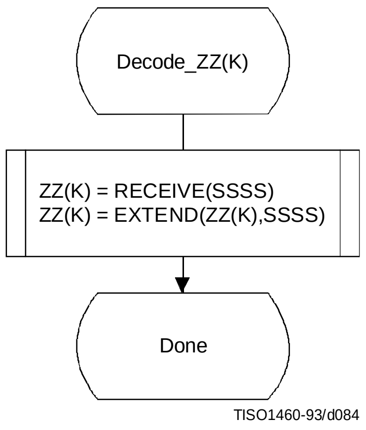


Figure F.14 – Decoding a non-zero AC coefficient

The decoding of the amplitude and sign of the non-zero coefficient is done in the procedure “Decode_ZZ(K)”, shown in Figure F.14.

DECODE is a procedure which returns the value, RS, associated with the next Huffman code in the code stream (see F.2.2.3). The values SSSS and R are derived from RS. The value of SSSS is the four low order bits of the composite value and R contains the value of RRRR (the four high order bits of the composite value). The interpretation of these values is described in F.1.2.2. EXTEND is shown in Figure F.12.

F.2.2.3 The DECODE procedure

The DECODE procedure decodes an 8-bit value which, for the DC coefficient, determines the difference magnitude category. For the AC coefficient this 8-bit value determines the zero run length and non-zero coefficient category.

Three tables, HUFFVAL, HUFFCODE, and HUFFSIZE, have been defined in Annex C. This particular implementation of DECODE makes use of the ordering of the Huffman codes in HUFFCODE according to both value and code size. Many other implementations of DECODE are possible.

NOTE – The values in HUFFVAL are assigned to each code in HUFFCODE and HUFFSIZE in sequence. There are no ordering requirements for the values in HUFFVAL which have assigned codes of the same length.

The implementation of DECODE described in this subclause uses three tables, MINCODE, MAXCODE and VALPTR, to decode a pointer to the HUFFVAL table. MINCODE, MAXCODE and VALPTR each have 16 entries, one for each possible code size. MINCODE(I) contains the smallest code value for a given length I, MAXCODE(I) contains the largest code value for a given length I, and VALPTR(I) contains the index to the start of the list of values in HUFFVAL which are decoded by code words of length I. The values in MINCODE and MAXCODE are signed 16-bit integers; therefore, a value of –1 sets all of the bits.

The procedure for generating these tables is shown in Figure F.15. The procedure for DECODE is shown in Figure F.16. Note that the 8-bit “VALUE” is returned to the procedure which invokes DECODE.

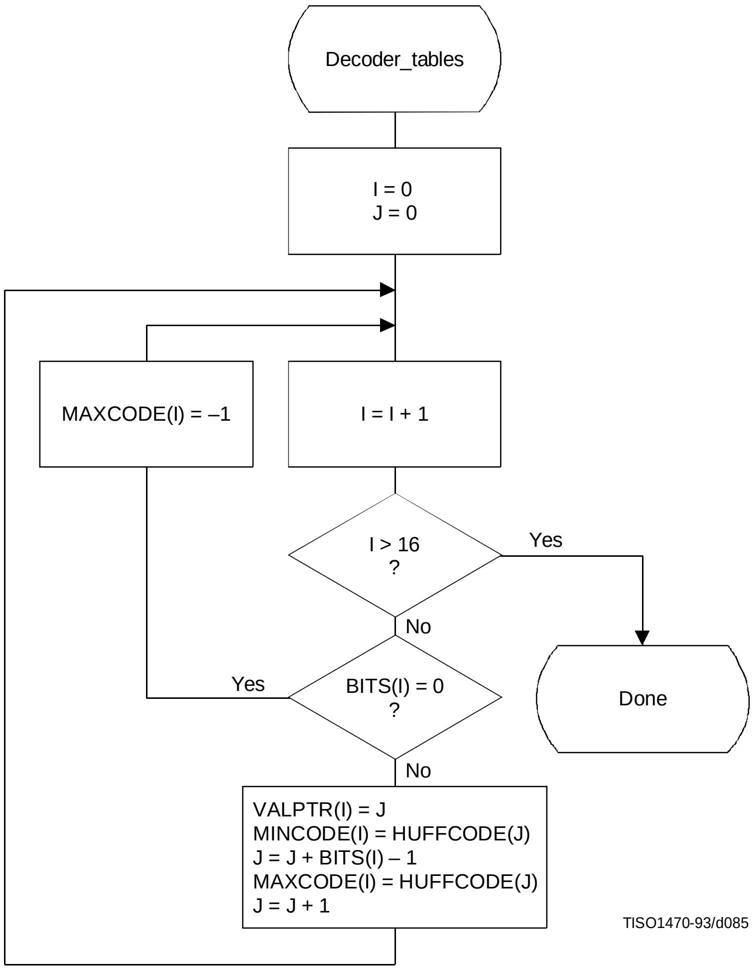


Figure F.15 – Decoder table generation

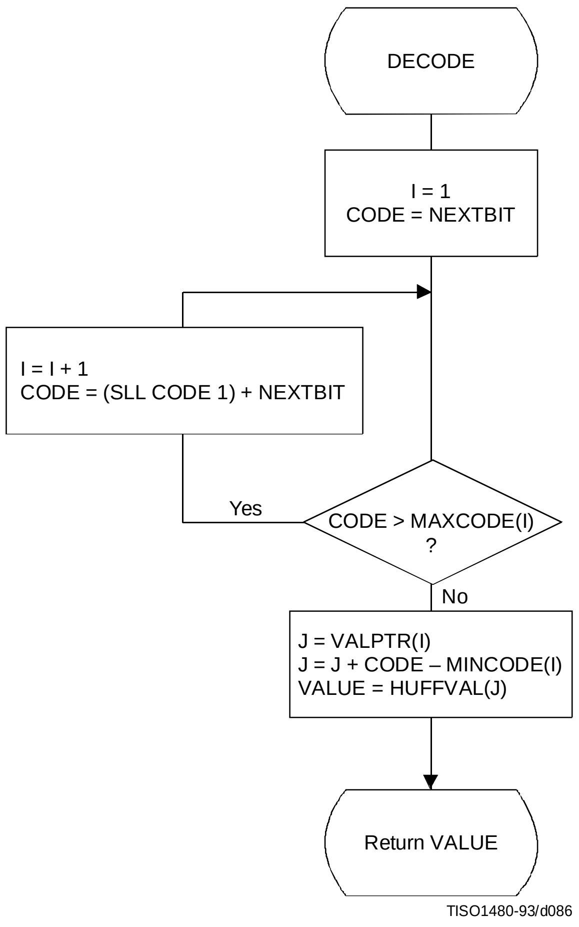


Figure F.16 – Procedure for DECODE

F.2.2.4 The RECEIVE procedure

RECEIVE(SSSS) is a procedure which places the next SSSS bits of the entropy-coded segment into the low order bits of DIFF, MSB first. It calls NEXTBIT and it returns the value of DIFF to the calling procedure (see Figure F.17).

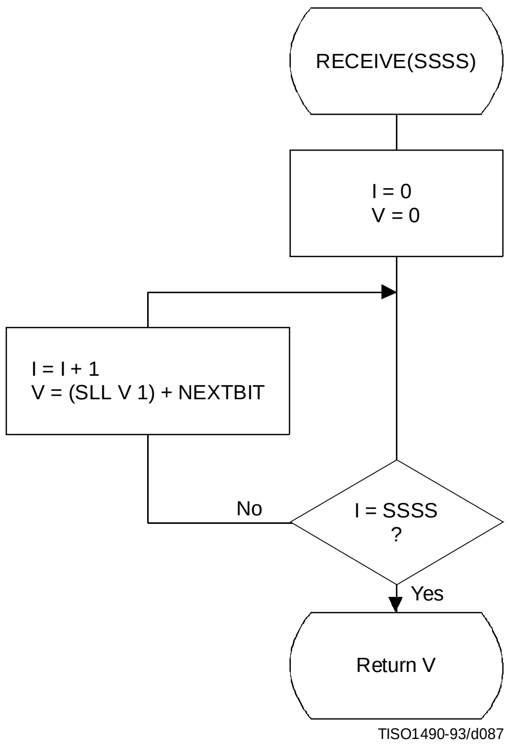


Figure F.17 – Procedure for RECEIVE(SSSS)

F.2.2.5 The NEXTBIT procedure

NEXTBIT reads the next bit of compressed data and passes it to higher level routines. It also intercepts and removes stuff bytes and detects markers. NEXTBIT reads the bits of a byte starting with the MSB (see Figure F.18).

Before starting the decoding of a scan, and after processing a RST marker, CNT is cleared. The compressed data are read one byte at a time, using the procedure NEXTBYTE. Each time a byte, B, is read, CNT is set to 8.

The only valid marker which may occur within the Huffman coded data is the RST_m marker. Other than the EOI or markers which may occur at or before the start of a scan, the only marker which can occur at the end of the scan is the DNL (define-number-of-lines).

Normally, the decoder will terminate the decoding at the end of the final restart interval before the terminating marker is intercepted. If the DNL marker is encountered, the current line count is set to the value specified by that marker. Since the DNL marker can only be used at the end of the first scan, the scan decode procedure must be terminated when it is encountered.

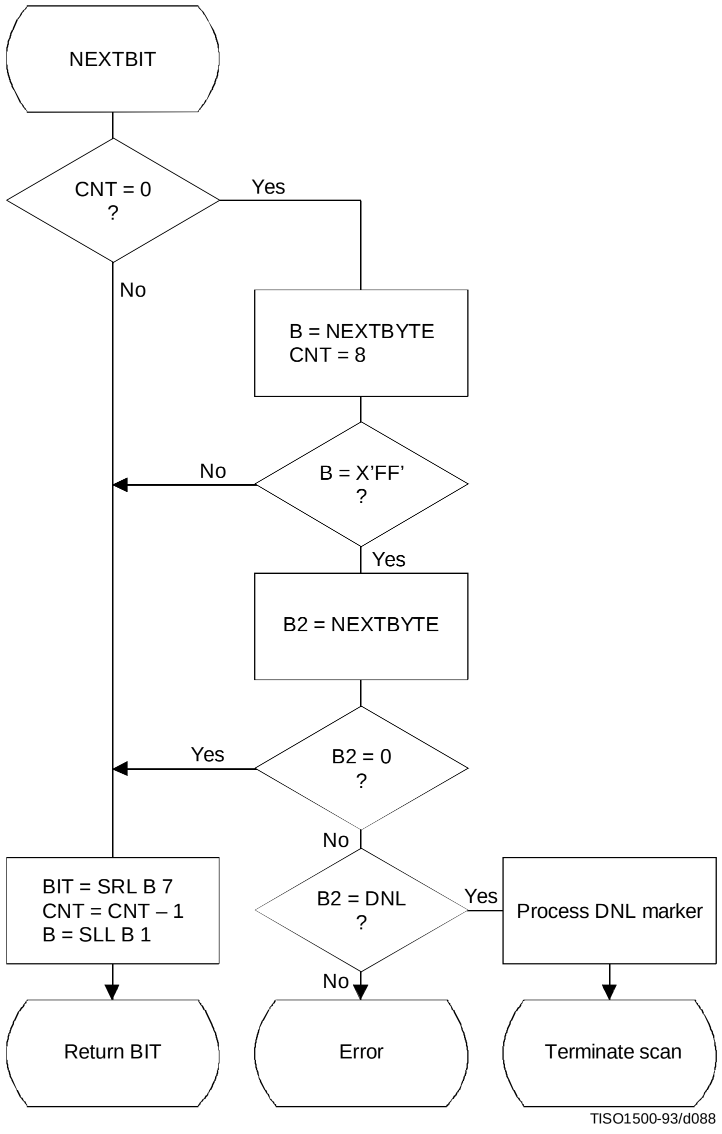


Figure F.18 – Procedure for fetching the next bit of compressed data

F.2.3 Sequential DCT decoding process with 8-bit precision extended to four sets of Huffman tables

This process is identical to the Baseline decoding process described in F.2.2, with the exception that the decoder shall be capable of using up to four DC and four AC Huffman tables within one scan. Four DC and four AC Huffman tables is the maximum allowed by this Specification.

F.2.4 Sequential DCT decoding process with arithmetic coding

This subclause describes the sequential DCT decoding process with arithmetic decoding.

The arithmetic decoding procedures for decoding binary decisions, initializing the statistical model, initializing the decoder, and resynchronizing the decoder are listed in Table D.4 of Annex D.

Some of the procedures in Table D.4 are used in the higher level control structure for scans and restart intervals described in F.2. At the beginning of scans and restart intervals, the probability estimates used in the arithmetic decoder are reset to the standard initial value as part of the Initdec procedure which restarts the arithmetic coder.

The statistical models defined in F.1.4.4 also apply to this decoding process.

The decoder shall be capable of using up to four DC and four AC conditioning tables and associated statistics areas within one scan.

F.2.4.1 Arithmetic decoding of DC coefficients

The basic structure of the decision sequence for decoding a DC difference value, DIFF, is shown in Figure F.19. The equivalent structure for the encoder is found in Figure F.4.

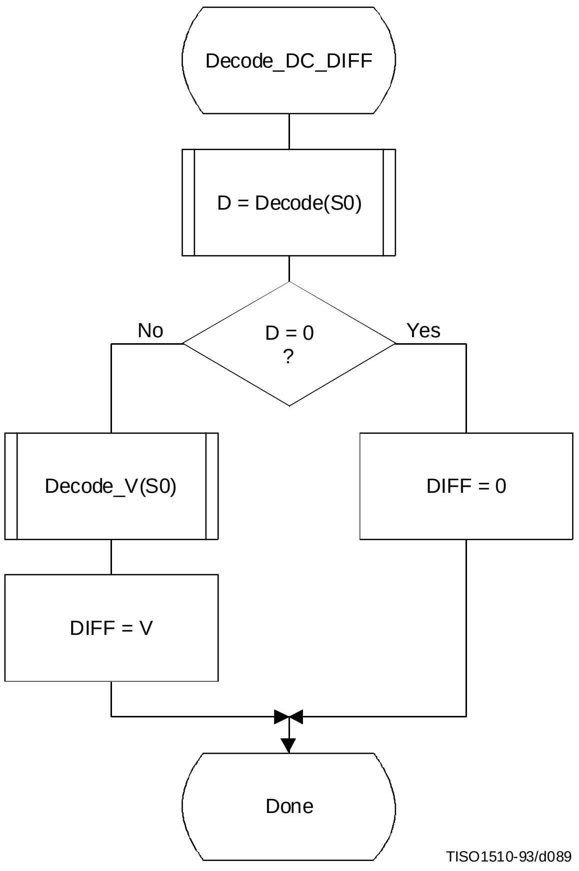


Figure F.19 – Arithmetic decoding of DC difference

The context-indices used in the DC decoding procedures are defined in Table F.4 (see F.1.4.4.1.3).

The "Decode" procedure returns the value "D" of the binary decision. If the value is not zero, the sign and magnitude of the non-zero DIFF must be decoded by the procedure "Decode_V(S0)".

F.2.4.2 Arithmetic Decoding of AC coefficients

The AC coefficients are decoded in the order that they occur in ZZ(1,...,63). The encoder procedure for the coding process is found in Figure F.5. Figure F.20 illustrates the decoding sequence.

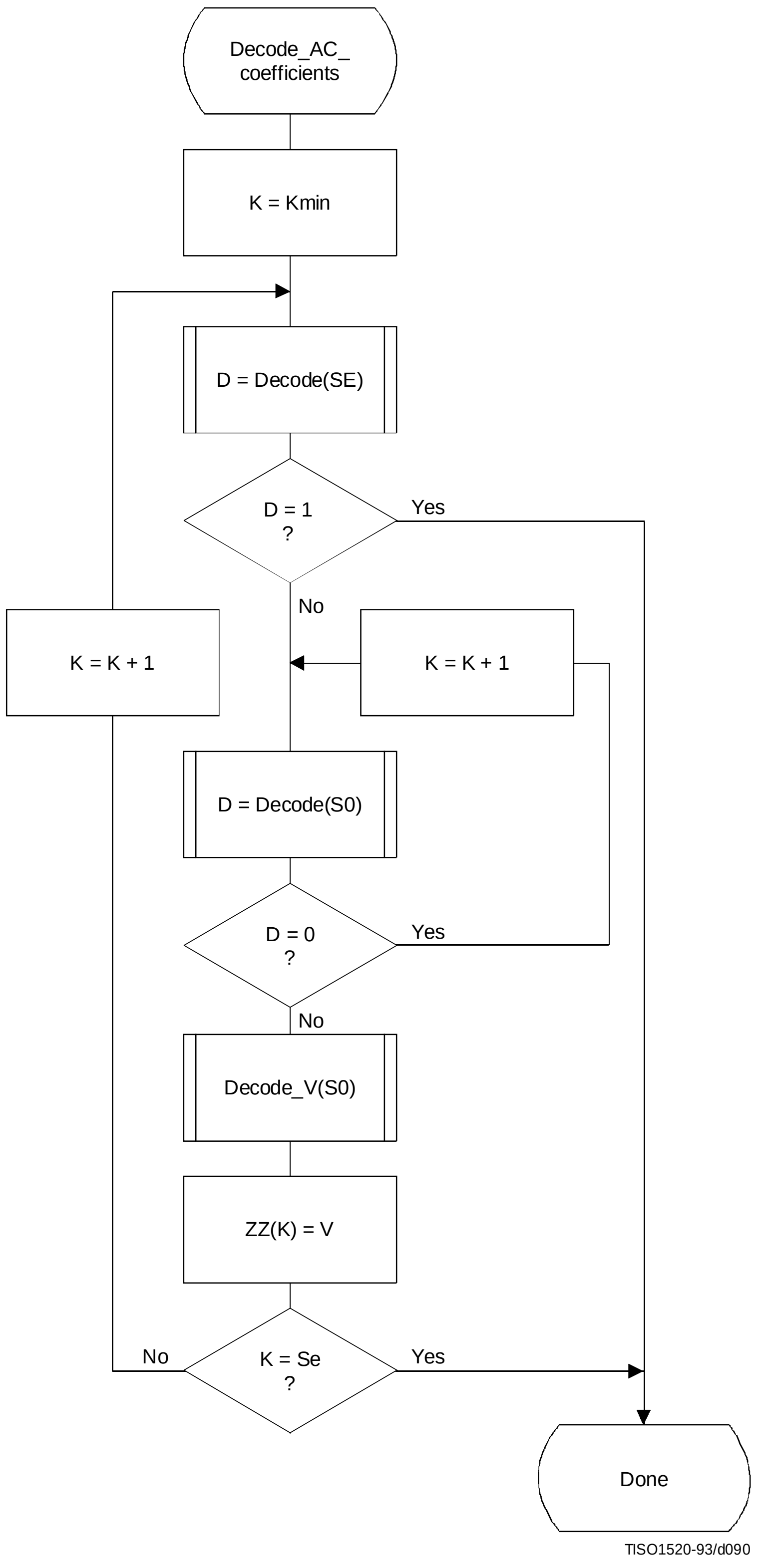


Figure F.20 – Procedure for decoding the AC coefficients

The context-indices used in the AC decoding procedures are defined in Table F.5 (see F.1.4.4.2).

In Figure F.20, K is the index to the zig-zag sequence position. For the sequential scan, Kmin = 1 and Se = 63. The decision at the top of the loop is the EOB decision. If the EOB occurs (D = 1), the remaining coefficients in the block are set to zero. The inner loop just below the EOB decoding decodes runs of zero coefficients. Whenever the coefficient is non-zero, “Decode_V” decodes the sign and magnitude of the coefficient. After each non-zero coefficient is decoded, the EOB decision is again decoded unless K = Se.

F.2.4.3 Decoding the binary decision sequence for non-zero DC differences and AC coefficients

Both the DC difference and the AC coefficients are represented as signed two’s complement 16-bit integer values. The decoding decision tree for these signed integer values is the same for both the DC and AC coding models. Note, however, that the statistical models are not the same.

F.2.4.3.1 Arithmetic decoding of non-zero values

Denoting either DC differences or AC coefficients as V, the non-zero signed integer value of V is decoded by the sequence shown in Figure F.21. This sequence first decodes the sign of V. It then decodes the magnitude category of V (Decode_log2_Sz), and then decodes the low order magnitude bits (Decode_Sz_bits). Note that the value decoded for Sz must be incremented by 1 to get the actual coefficient magnitude.

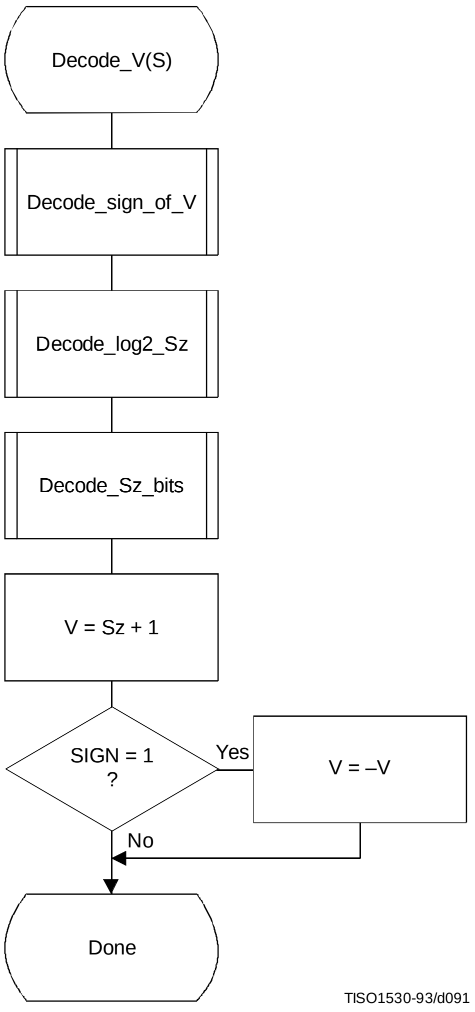


Figure F.21 – Sequence of procedures in decoding non-zero values of V

F.2.4.3.1.1 Decoding the sign

The sign is decoded by the procedure shown in Figure F.22.

The context-indices are defined for DC decoding in Table F.4 and AC decoding in Table F.5.

If SIGN = 0, the sign of the coefficient is positive; if SIGN = 1, the sign of the coefficient is negative.

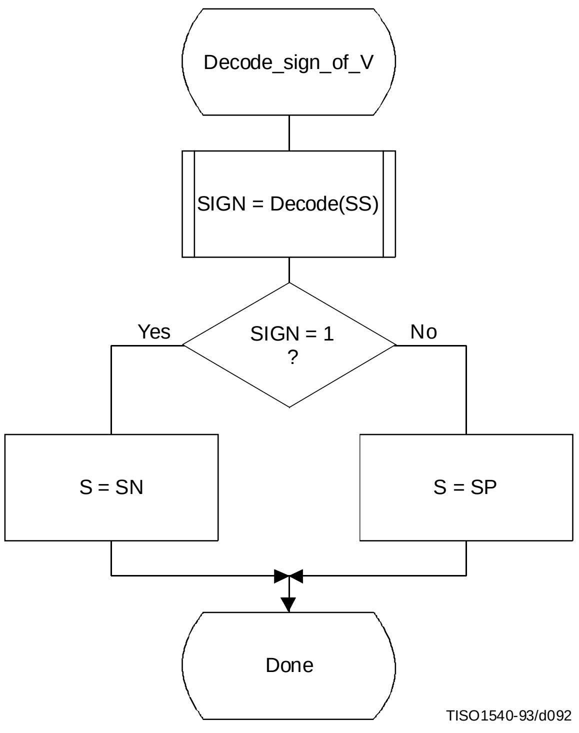


Figure F.22 – Decoding the sign of V

F.2.4.3.1.2 Decoding the magnitude category

The context-index S is set in Decode_sign_of_V and the context-index values X1 and X2 are defined for DC coding in Table F.4 and for AC coding in Table F.5.

In Figure F.23, M is set to the upper bound for the magnitude and shifted left until the decoded decision is zero. It is then shifted right by 1 to become the leading bit of the magnitude of Sz.

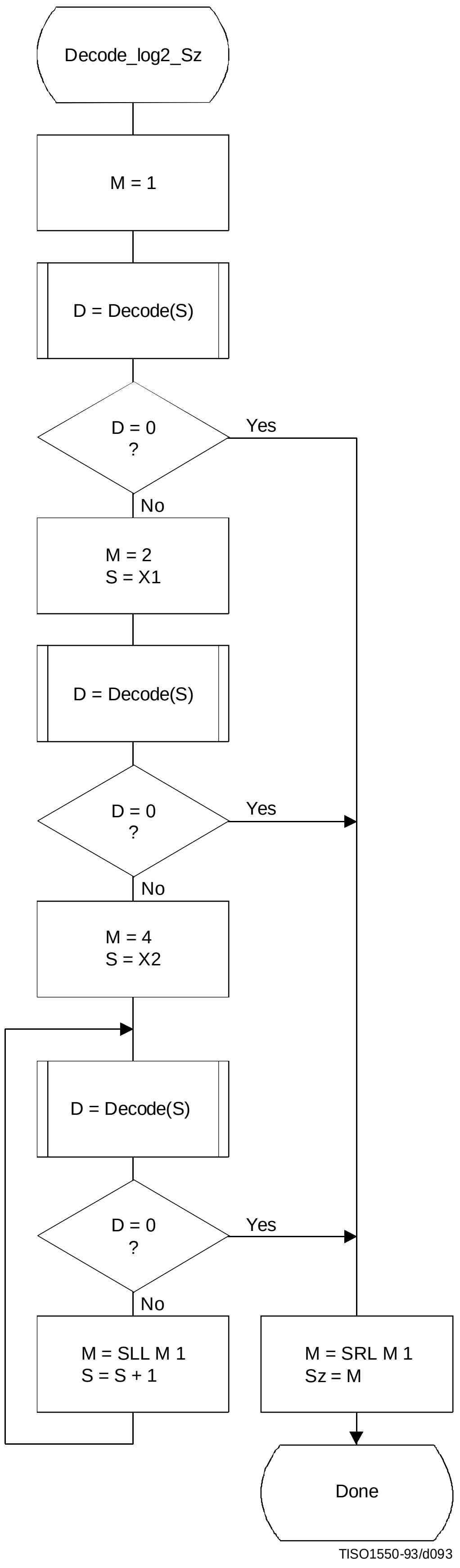


Figure F.23 – Decoding procedure to establish the magnitude category

F.2.4.3.1.3 Decoding the exact value of the magnitude

After the magnitude category is decoded, the low order magnitude bits are decoded. These bits are decoded in order of decreasing bit significance. The procedure is shown in Figure F.24.

The context-index S is set in Decode_log2_Sz.

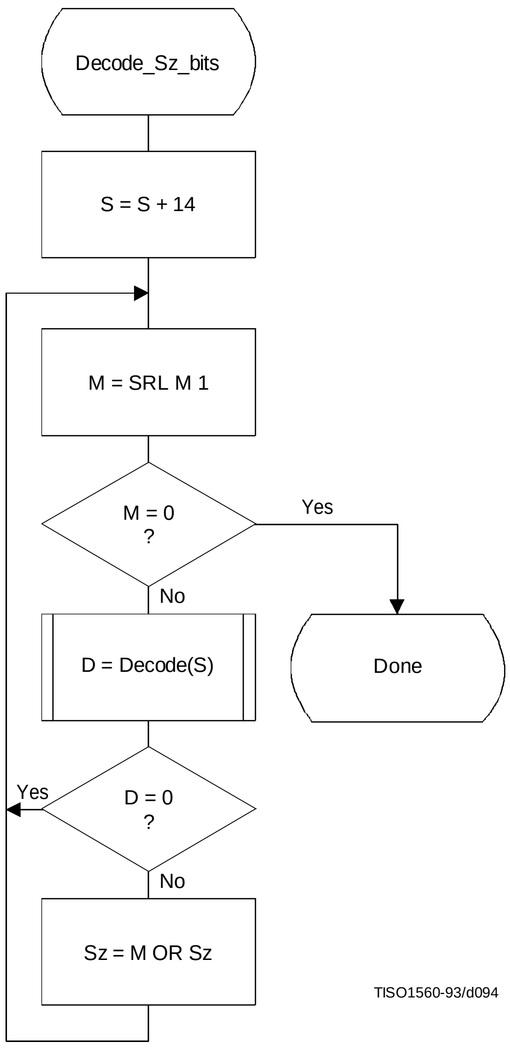


Figure F.24 – Decision sequence to decode the magnitude bit pattern

F.2.4.4 Decoder restart

The RST_m markers which are added to the compressed data between each restart interval have a two byte value which cannot be generated by the coding procedures. These two byte sequences can be located without decoding, and can therefore be used to resynchronize the decoder. RST_m markers can therefore be used for error recovery.

Before error recovery procedures can be invoked, the error condition must first be detected. Errors during decoding can show up in two places:

a) The decoder fails to find the expected marker at the point where it is expecting resynchronization.

b) Physically impossible data are decoded. For example, decoding a magnitude beyond the range of values allowed by the model is quite likely when the compressed data are corrupted by errors. For arithmetic decoders this error condition is extremely important to detect, as otherwise the decoder may reach a condition where it uses the compressed data very slowly.

   NOTE – Some errors will not cause the decoder to lose synchronization. In addition, recovery is not possible for all errors; for example, errors in the headers are likely to be catastrophic. The two error conditions listed above, however, almost always cause the decoder to lose synchronization in a way which permits recovery.

In regaining synchronization, the decoder can make use of the modulo 8 coding restart interval number in the low order bits of the RST_m marker. By comparing the expected restart interval number to the value in the next RST_m marker in the compressed image data, the decoder can usually recover synchronization. It then fills in missing lines in the output data by replication or some other suitable procedure, and continues decoding. Of course, the reconstructed image will usually be highly corrupted for at least a part of the restart interval where the error occurred.

F.2.5 Sequential DCT decoding process with Huffman coding and 12-bit precision

This process is identical to the sequential DCT process defined for 8-bit sample precision and extended to four Huffman tables, as documented in F.2.3, but with the following changes.

F.2.5.1 Structure of DC Huffman decode table

The general structure of the DC Huffman decode table is extended as described in F.1.5.1.

F.2.5.2 Structure of AC Huffman decode table

The general structure of the AC Huffman decode table is extended as described in F.1.5.2.

F.2.6 Sequential DCT decoding process with arithmetic coding and 12-bit precision

The process is identical to the sequential DCT process for 8-bit precision except for changes in the precision of the IDCT computation.

The structure of the decoding procedure in F.2.4 is already defined for a 12-bit input precision.

Annex G

Progressive DCT-based mode of operation

(This annex forms an integral part of this Recommendation | International Standard)

This annex provides a **functional specification** of the following coding processes for the progressive DCT-based mode of operation:

1) spectral selection only, Huffman coding, 8-bit sample precision;
2) spectral selection only, arithmetic coding, 8-bit sample precision;
3) full progression, Huffman coding, 8-bit sample precision;
4) full progression, arithmetic coding, 8-bit sample precision;
5) spectral selection only, Huffman coding, 12-bit sample precision;
6) spectral selection only, arithmetic coding, 12-bit sample precision;
7) full progression, Huffman coding, 12-bit sample precision;
8) full progression, arithmetic coding, 12-bit sample precision.

For each of these, the encoding process is specified in G.1, and the decoding process is specified in G.2. The functional specification is presented by means of specific flow charts for the various procedures which comprise these coding processes.

NOTE – There is **no requirement** in this Specification that any encoder or decoder which embodies one of the above-named processes shall implement the procedures in precisely the manner specified by the flow charts in this annex. It is necessary only that an encoder or decoder implement the **function** specified in this annex. The sole criterion for an encoder or decoder to be considered in compliance with this Specification is that it satisfy the requirements given in clause 6 (for encoders) or clause 7 (for decoders), as determined by the compliance tests specified in Part 2.

The number of Huffman or arithmetic conditioning tables which may be used within the same scan is four.

Two complementary progressive procedures are defined, spectral selection and successive approximation.

In spectral selection the DCT coefficients of each block are segmented into frequency bands. The bands are coded in separate scans.

In successive approximation the DCT coefficients are divided by a power of two before coding. In the decoder the coefficients are multiplied by that same power of two before computing the IDCT. In the succeeding scans the precision of the coefficients is increased by one bit in each scan until full precision is reached.

An encoder or decoder implementing a full progression uses spectral selection within successive approximation. An allowed subset is spectral selection alone.

Figure G.1 illustrates the spectral selection and successive approximation progressive processes.

G.1 Progressive DCT-based encoding processes

G.1.1 Control procedures and coding models for progressive DCT-based procedures

G.1.1.1 Control procedures for progressive DCT-based encoders

The control procedures for encoding an image and its constituent parts – the frame, scan, restart interval and MCU – are given in Figures E.1 through E.5.

The control structure for encoding a frame is the same as for the sequential procedures. However, it is convenient to calculate the FDCT for the entire set of components in a frame before starting the scans. A buffer which is large enough to store all of the DCT coefficients may be used for this progressive mode of operation.

The number of scans is determined by the progression defined; the number of scans may be much larger than the number of components in the frame.

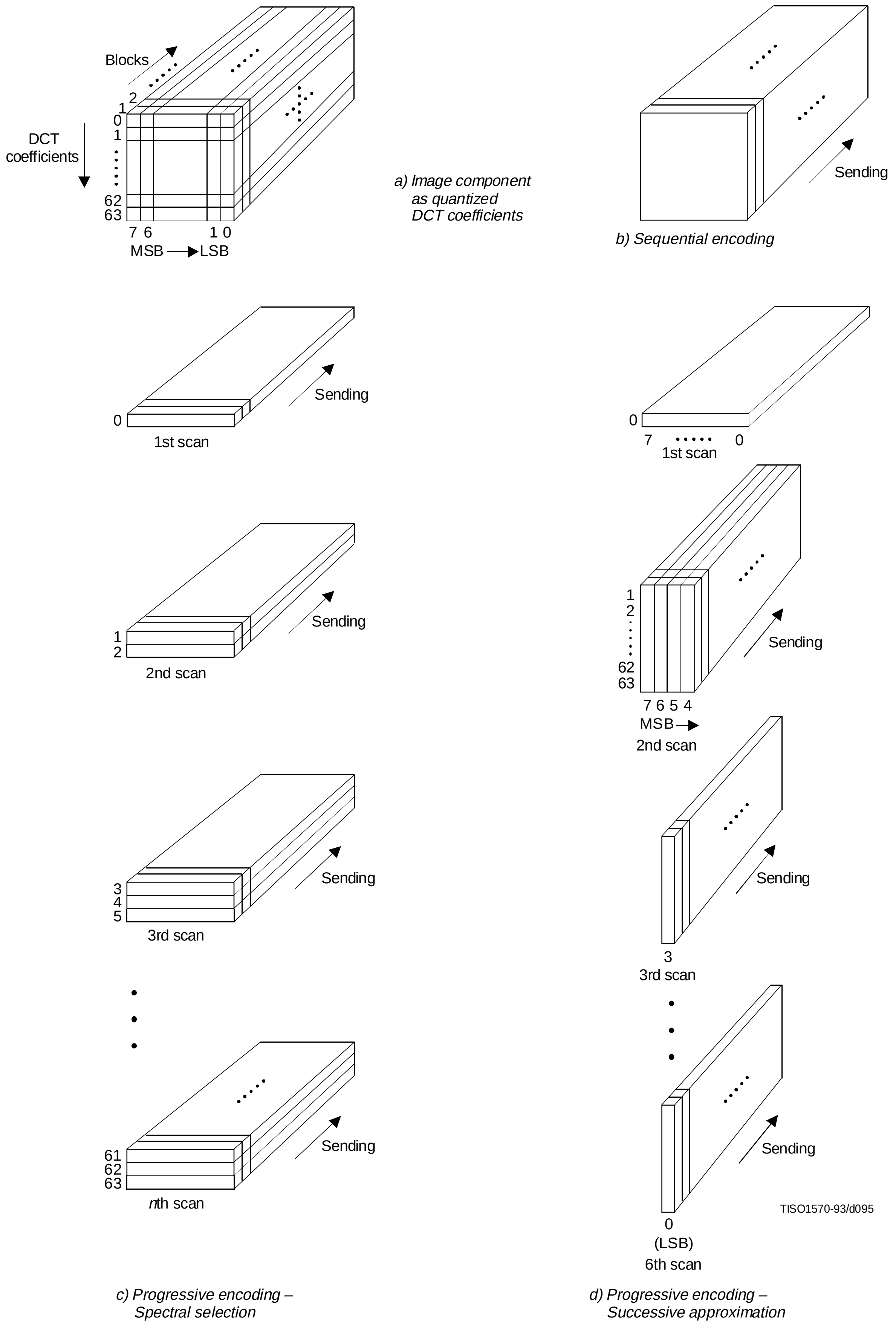


Figure G.1 – Spectral selection and successive approximation progressive processes

The procedure for encoding a MCU (see Figure E.5) repetitively invokes the procedure for coding a data unit. For DCT-based encoders the data unit is an 8×8 block of samples.

Only a portion of each 8×8 block is coded in each scan, the portion being determined by the scan header parameters Ss, Se, Ah, and Al (see B.2.3). The procedures used to code portions of each 8×8 block are described in this annex. Note, however, that where these procedures are identical to those used in the sequential DCT-based mode of operation, the sequential procedures are simply referenced.

G.1.1.1.1 Spectral selection control

In spectral selection the zig-zag sequence of DCT coefficients is segmented into bands. A band is defined in the scan header by specifying the starting and ending indices in the zig-zag sequence. One band is coded in a given scan of the progression. DC coefficients are always coded separately from AC coefficients, and only scans which code DC coefficients may have interleaved blocks from more than one component. All other scans shall have only one component. With the exception of the first DC scans for the components, the sequence of bands defined in the scans need not follow the zig-zag ordering. For each component, a first DC scan shall precede any AC scans.

G.1.1.1.2 Successive approximation control

If successive approximation is used, the DCT coefficients are reduced in precision by the point transform (see A.4) defined in the scan header (see B.2.3). The successive approximation bit position parameter Al specifies the actual point transform, and the high four bits (Ah) – if there are preceding scans for the band – contain the value of the point transform used in those preceding scans. If there are no preceding scans for the band, Ah is zero.

Each scan which follows the first scan for a given band progressively improves the precision of the coefficients by one bit, until full precision is reached.

G.1.1.2 Coding models for progressive DCT-based encoders

If successive approximation is used, the DCT coefficients are reduced in precision by the point transform (see A.4) defined in the scan header (see B.2.3). These models also apply to the progressive DCT-based encoders, but with the following changes.

G.1.1.2.1 Progressive encoding model for DC coefficients

If Al is not zero, the point transform for DC coefficients shall be used to reduce the precision of the DC coefficients. If Ah is zero, the coefficient values (as modified by the point transform) shall be coded, using the procedure described in Annex F. If Ah is not zero, the least significant bit of the point transformed DC coefficients shall be coded, using the procedures described in this annex.

G.1.1.2.2 Progressive encoding model for AC coefficients

If Al is not zero, the point transform for AC coefficients shall be used to reduce the precision of the AC coefficients. If Ah is zero, the coefficient values (as modified by the point transform) shall be coded using modifications of the procedures described in Annex F. These modifications are described in this annex. If Ah is not zero, the precision of the coefficients shall be improved using the procedures described in this annex.

G.1.2 Progressive encoding procedures with Huffman coding

G.1.2.1 Progressive encoding of DC coefficients with Huffman coding

The first scan for a given component shall encode the DC coefficient values using the procedures described in F.1.2.1. If the successive approximation bit position parameter Al is not zero, the coefficient values shall be reduced in precision by the point transform described in Annex A before coding.

In subsequent scans using successive approximation the least significant bits are appended to the compressed bit stream without compression or modification (see G.1.2.3), except for byte stuffing.

G.1.2.2 Progressive encoding of AC coefficients with Huffman coding

In spectral selection and in the first scan of successive approximation for a component, the AC coefficient coding model is similar to that used by the sequential procedures. However, the Huffman code tables are extended to include coding of runs of End-Of-Bands (EOBs). See Table G.1.

Table G.1 – EOBn code run length extensions

EOBn code	Run length
EOB0	1
EOB1	2,3
EOB2	4..7
EOB3	8..15
EOB4	16..31
EOB5	32..63
EOB6	64..127
EOB7	128..255
EOB8	256..511
EOB9	512..1 023
EOB10	1 024..2 047
EOB11	2 048..4 095
EOB12	4 096..8 191
EOB13	8 192..16 383
EOB14	16 384..32 767

The end-of-band run structure allows efficient coding of blocks which have only zero coefficients. An EOB run of length 5 means that the current block and the next four blocks have an end-of-band with no intervening non-zero coefficients. The EOB run length is limited only by the restart interval.

The extension of the code table is illustrated in Figure G.2.

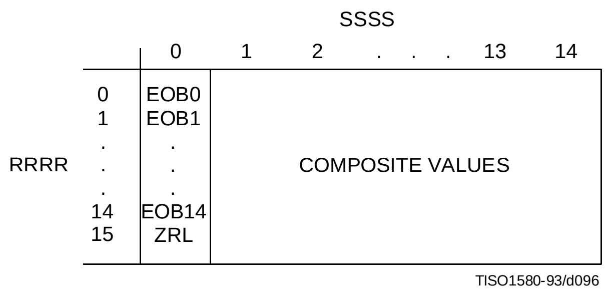


Figure G.2 – Two-dimensional value array for Huffman coding

The EOBn code sequence is defined as follows. Each EOBn code is followed by an extension field similar to the extension field for the coefficient amplitudes (but with positive numbers only). The number of bits appended to the EOBn code is the minimum number required to specify the run length.

If an EOB run is greater than 32 767, it is coded as a sequence of EOB runs of length 32 767 followed by a final EOB run sufficient to complete the run.

At the beginning of each restart interval the EOB run count, EOBRUN, is set to zero. At the end of each restart interval any remaining EOB run is coded.

The Huffman encoding procedure for AC coefficients in spectral selection and in the first scan of successive approximation is illustrated in Figures G.3, G.4, G.5, and G.6.

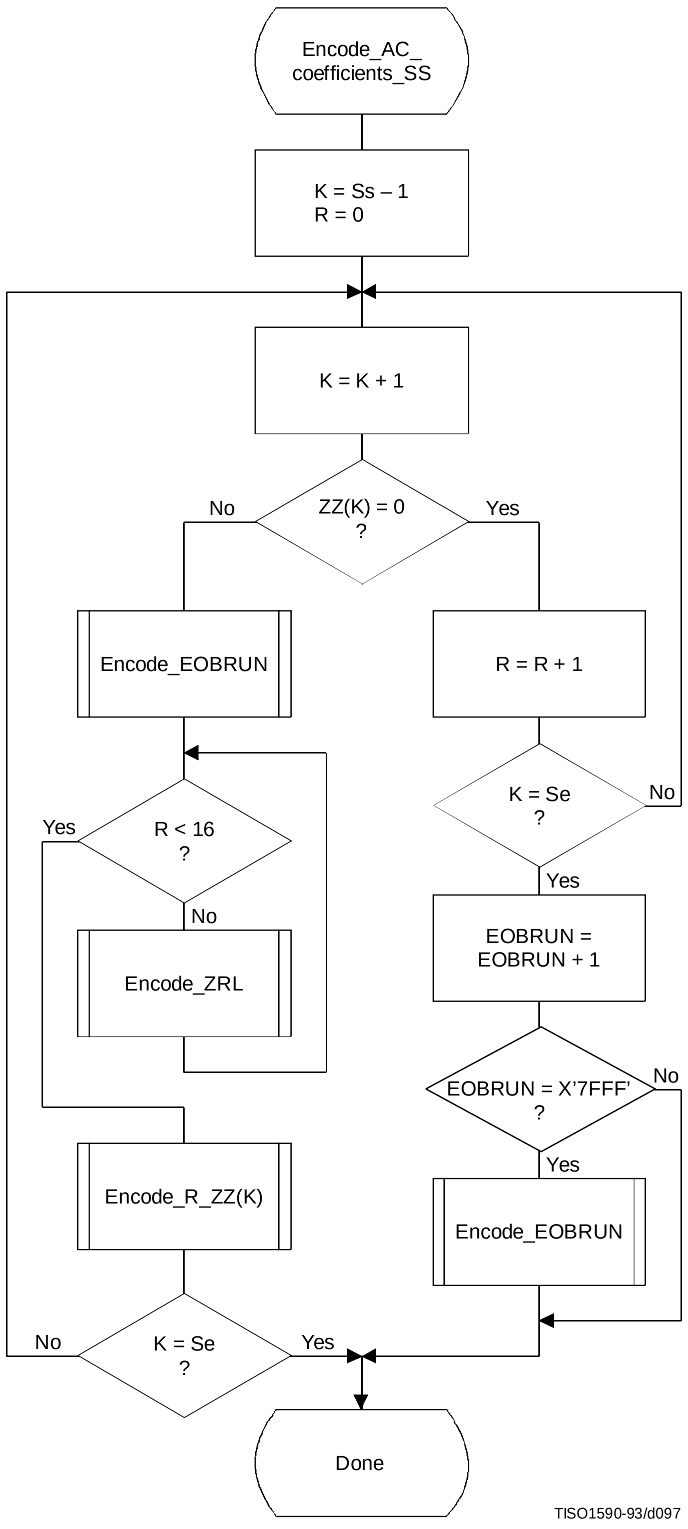


Figure G.3 – Procedure for progressive encoding of AC coefficients with Huffman coding

In Figure G.3, Ss is the start of spectral selection, Se is the end of spectral selection, K is the index into the list of coefficients stored in the zig-zag sequence ZZ, R is the run length of zero coefficients, and EOBRUN is the run length of EOBs. EOBRUN is set to zero at the start of each restart interval.

If the scan header parameter Al (successive approximation bit position low) is not zero, the DCT coefficient values ZZ(K) in Figure G.3 and figures which follow in this annex, including those in the arithmetic coding section, shall be replaced by the point transformed values ZZ'(K), where ZZ'(K) is defined by:

$$ZZ'(K) = \frac{ZZ(K)x}{2^{Al}}$$

EOBSIZE is a procedure which returns the size of the EOB extension field given the EOB run length as input. CSIZE is a procedure which maps an AC coefficient to the SSSS value defined in the subclauses on sequential encoding (see F.1.1 and F.1.3).

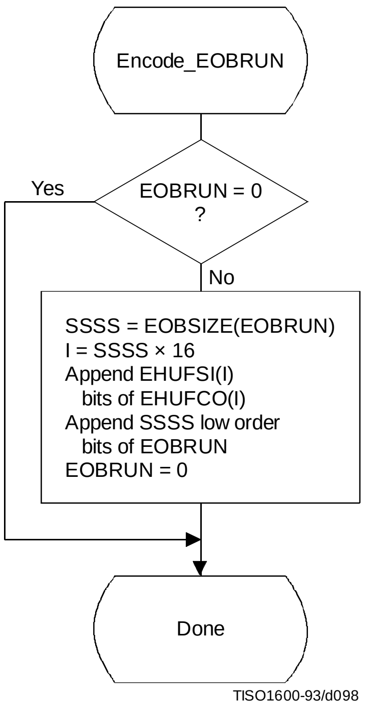


Figure G.4 – Progressive encoding of a non-zero AC coefficient

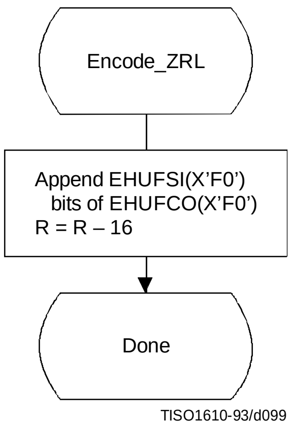


Figure G.5 – Encoding of the run of zero coefficients

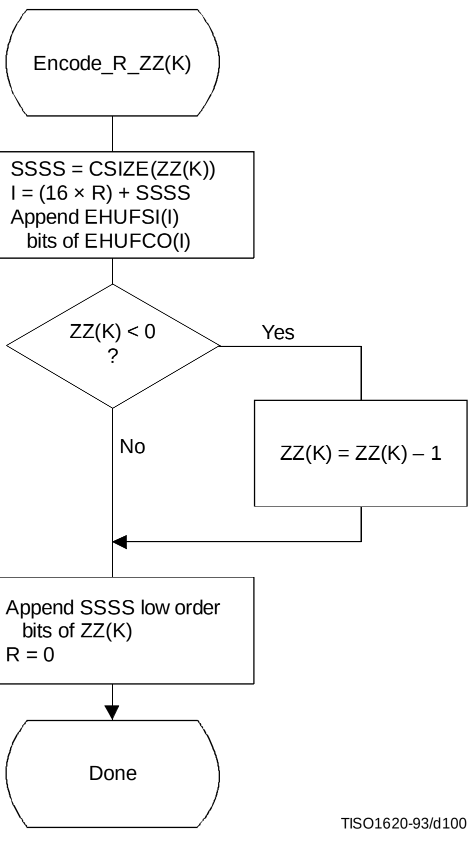


Figure G.6 – Encoding of the zero run and non-zero coefficient

G.1.2.3 Coding model for subsequent scans of successive approximation

The Huffman coding structure of the subsequent scans of successive approximation for a given component is similar to the coding structure of the first scan of that component.

The structure of the AC code table is identical to the structure described in G.1.2.2. Each non-zero point transformed coefficient that has a zero history (i.e. that has a value ± 1, and therefore has not been coded in a previous scan) is defined by a composite 8-bit run length-magnitude value of the form:

RRRRSSSS

The four most significant bits, RRRR, give the number of zero coefficients that are between the current coefficient and the previously coded coefficient (or the start of band). Coefficients with non-zero history (a non-zero value coded in a previous scan) are skipped over when counting the zero coefficients. The four least significant bits, SSSS, provide the magnitude category of the non-zero coefficient; for a given component the value of SSSS can only be one.

The run length-magnitude composite value is Huffman coded and each Huffman code is followed by additional bits:

a) One bit codes the sign of the newly non-zero coefficient. A 0-bit codes a negative sign; a 1-bit codes a positive sign.

b) For each coefficient with a non-zero history, one bit is used to code the correction. A 0-bit means no correction and a 1-bit means that one shall be added to the (scaled) decoded magnitude of the coefficient.

Non-zero coefficients with zero history are coded with a composite code of the form:

HUFFCO(RRRRSSSS) + additional bit (rule a) + correction bits (rule b)

In addition whenever zero runs are coded with ZRL or EOBn codes, correction bits for those coefficients with non-zero history contained within the zero run are appended according to rule b above.

For the Huffman coding version of Encode_AC_Coefficients_SA the EOB is defined to be the position of the last point transformed coefficient of magnitude 1 in the band. If there are no coefficients of magnitude 1, the EOB is defined to be zero.

NOTE – The definition of EOB is different for Huffman and arithmetic coding procedures.

In Figures G.7 and G.8 BE is the count of buffered correction bits at the start of coding of the block. BE is initialized to zero at the start of each restart interval. At the end of each restart interval any remaining buffered bits are appended to the bit stream following the last EOBn Huffman code and associated appended bits.

In Figures G.7 and G.9, BR is the count of buffered correction bits which are appended to the bit stream according to rule b. BR is set to zero at the beginning of each Encode_AC_Coefficients_SA. At the end of each restart interval any remaining buffered bits are appended to the bit stream following the last Huffman code and associated appended bits.

G.1.3 Progressive encoding procedures with arithmetic coding

G.1.3.1 Progressive encoding of DC coefficients with arithmetic coding

The first scan for a given component shall encode the DC coefficient values using the procedures described in F.1.4.1. If the successive approximation bit position parameter is not zero, the coefficient values shall be reduced in precision by the point transform described in Annex A before coding.

In subsequent scans using successive approximation the least significant bits shall be coded as binary decisions using a fixed probability estimate of 0.5 (Qe = X’5A1D’, MPS = 0).

G.1.3.2 Progressive encoding of AC coefficients with arithmetic coding

Except for the point transform scaling of the DCT coefficients and the grouping of the coefficients into bands, the first scan(s) of successive approximation is identical to the sequential encoding procedure described in F.1.4. If Kmin is equated to Ss, the index of the first AC coefficient index in the band, the flow chart shown in Figure F.5 applies. The EOB decision in that figure refers to the “end-of-band” rather than the “end-of-block”. For the arithmetic coding version of Encode_AC_Coefficients_SA (and all other AC coefficient coding procedures) the EOB is defined to be the position following the last non-zero coefficient in the band.

NOTE - The definition of EOB is different for Huffman and arithmetic coding procedures.

The statistical model described in F.1.4 also holds. For this model the default value of Kx is 5. Other values of Kx may be specified using the DAC marker code (Annex B). The following calculation for Kx has proven to give good results for 8-bit precision samples:

$$Kx = Kmin + SRL\ \ (8 + Se - Kmin)\ \ 4$$

This expression reduces to the default of Kx = 5 when the band is from index 1 to index 63.

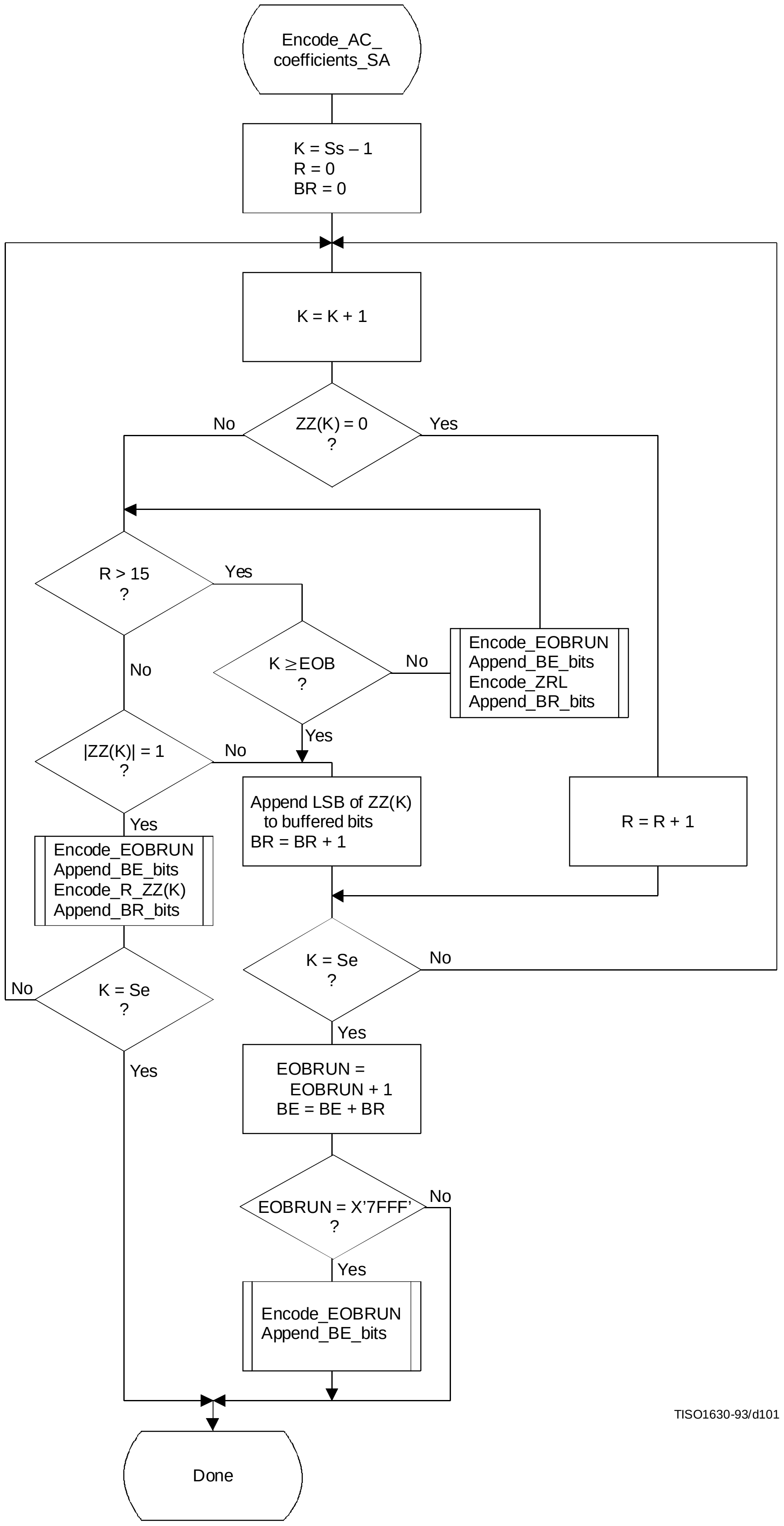


Figure G.7 – Successive approximation coding of AC coefficients using Huffman coding

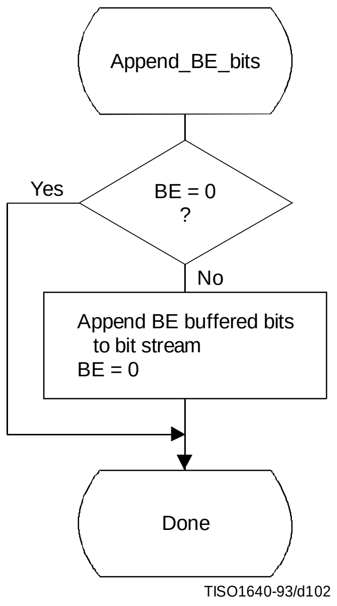


Figure G.8 – Transferring BE buffered bits from buffer to bit stream

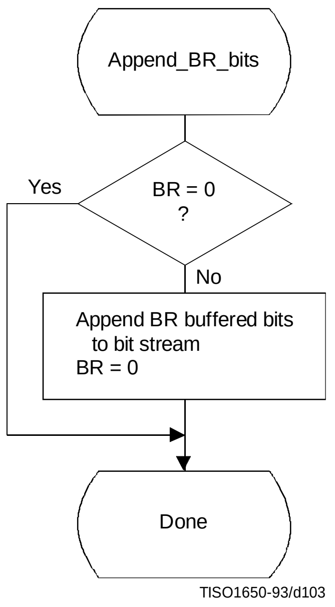


Figure G.9 – Transferring BR buffered bits from buffer to bit stream

G.1.3.3 Coding model for subsequent scans of successive approximation

The procedure "Encode_AC_Coefficient_SA" shown in Figure G.10 increases the precision of the AC coefficient values in the band by one bit.

As in the first scan of successive approximation for a component, an EOB decision is coded at the start of the band and after each non-zero coefficient.

However, since the end-of-band index of the previous successive approximation scan for a given component, EOBx, is known from the data coded in the prior scan of that component, this decision is bypassed whenever the current index, K, is less than EOBx. As in the first scan(s), the EOB decision is also bypassed whenever the last coefficient in the band is not zero. The decision ZZ(K) = 0 decodes runs of zero coefficients. If the decoder is at this step of the procedure, at least one non-zero coefficient remains in the band of the block being coded. If ZZ(K) is not zero, the procedure in Figure G.11 is followed to code the value.

The context-indices in Figures G.10 and G.11 are defined in Table G.2 (see G.1.3.3.1). The signs of coefficients with magnitude of one are coded with a fixed probability value of approximately 0.5 (Qe = X'5A1D', MPS = 0).

G.1.3.3.1 Statistical model for subsequent successive approximation scans

As shown in Table G.2, each statistics area for subsequent successive approximation scans of AC coefficients consists of a contiguous set of 189 statistics bins. The signs of coefficients with magnitude of one are coded with a fixed probability value of approximately 0.5 (Qe = X'5A1D', MPS = 0).

G.2 Progressive decoding of the DCT

The description of the computation of the IDCT and the dequantization procedure contained in A.3.3 and A.3.4 apply to the progressive operation.

Progressive decoding processes must be able to decompress compressed image data which requires up to four sets of Huffman or arithmetic coder conditioning tables within a scan.

In order to avoid repetition, detailed flow diagrams of progressive decoder operation are not included. Decoder operation is defined by reversing the function of each step described in the encoder flow charts, and performing the steps in reverse order.

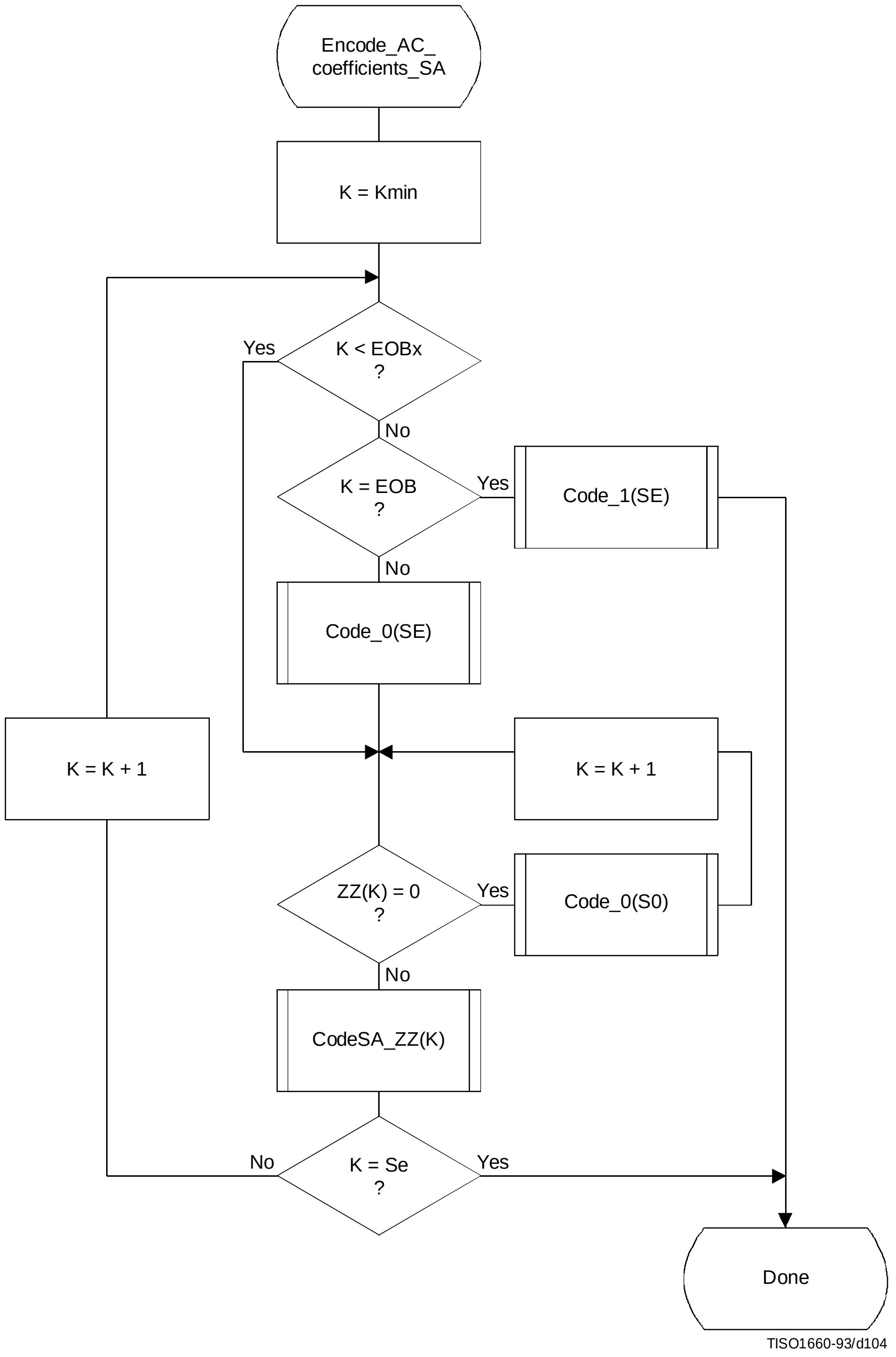


Figure G.10 – Subsequent successive approximation scans for coding of AC coefficients using arithmetic coding

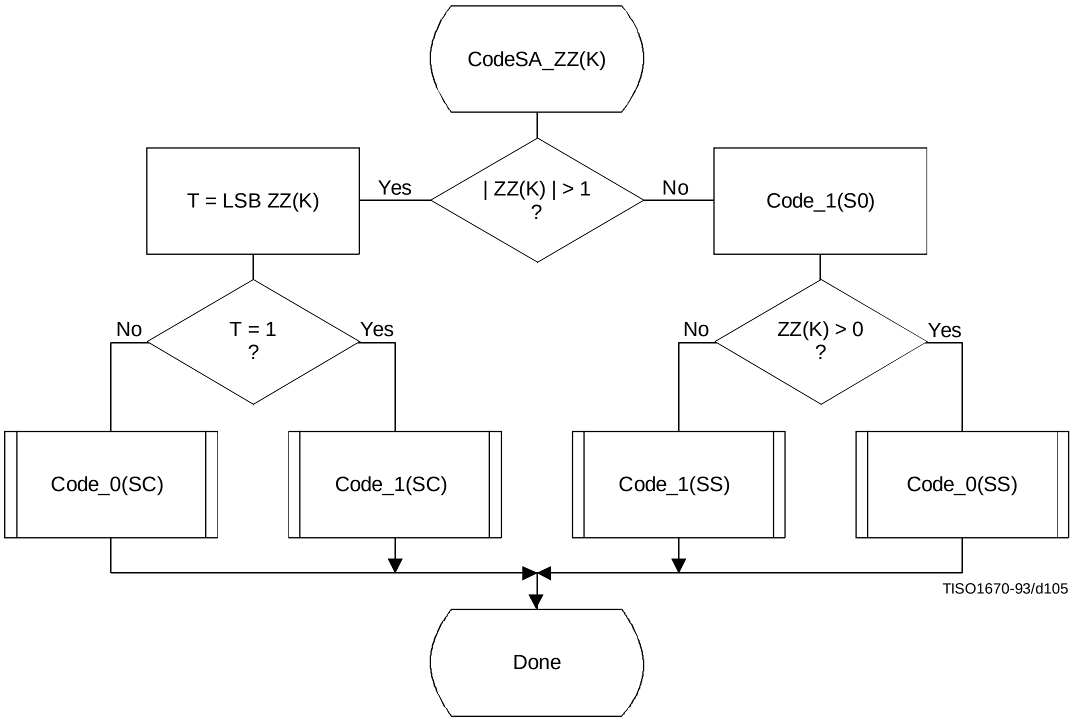


Figure G.11 – Coding non-zero coefficients for subsequent successive approximation scans

Table G.2 – Statistical model for subsequent scans of successive approximation coding of AC coefficient

Context-index	AC coding	Coding decision
SE	3 × (K–1)	K = EOB
S0	SE + 1	V = 0
SS	Fixed estimate	Sign
SC	S0 + 1	LSB ZZ(K) = 1

Annex H

Lossless mode of operation

(This annex forms an integral part of this Recommendation | International Standard)

This annex provides a **functional specification** of the following coding processes for the lossless mode of operation:

1) lossless processes with Huffman coding;

2) lossless processes with arithmetic coding.

For each of these, the encoding process is specified in H.1, and the decoding process is specified in H.2. The functional specification is presented by means of specific procedures which comprise these coding processes.

NOTE – There is **no requirement** in this Specification that any encoder or decoder which embodies one of the above-named processes shall implement the procedures in precisely the manner specified in this annex. It is necessary only that an encoder or decoder implement the **function** specified in this annex. The sole criterion for an encoder or decoder to be considered in compliance with this Specification is that it satisfy the requirements given in clause 6 (for encoders) or clause 7 (for decoders), as determined by the compliance tests specified in Part 2.

The processes which provide for sequential lossless encoding and decoding are not based on the DCT. The processes used are spatial processes based on the coding model developed for the DC coefficients of the DCT. However, the model is extended by incorporating a set of selectable one- and two-dimensional predictors, and for interleaved data the ordering of samples for the one-dimensional predictor can be different from that used in the DCT-based processes.

Either Huffman coding or arithmetic coding entropy coding may be employed for these lossless encoding and decoding processes. The Huffman code table structure is extended to allow up to 16-bit precision for the input data. The arithmetic coder statistical model is extended to a two-dimensional form.

H.1 Lossless encoder processes

H.1.1 Lossless encoder control procedures

Subclause E.1 contains the encoder control procedures. In applying these procedures to the lossless encoder, the data unit is one sample.

Input data precision may be from 2 to 16 bits/sample. If the input data path has different precision from the input data, the data shall be aligned with the least significant bits of the input data path. Input data is represented as unsigned integers and is not level shifted prior to coding.

When the encoder is reset in the restart interval control procedure (see E.1.4), the prediction is reset to a default value. If arithmetic coding is used, the statistics are also reset.

For the lossless processes the restart interval shall be an integer multiple of the number of MCU in an MCU-row.

H.1.2 Coding model for lossless encoding

The coding model developed for encoding the DC coefficients of the DCT is extended to allow a selection from a set of seven one-dimensional and two-dimensional predictors. The predictor is selected in the scan header (see Annex B). The same predictor is used for all components of the scan. Each component in the scan is modeled independently, using predictions derived from neighbouring samples of that component.

H.1.2.1 Prediction

Figure H.1 shows the relationship between the positions (a, b, c) of the reconstructed neighboring samples used for prediction and the position of x, the sample being coded.

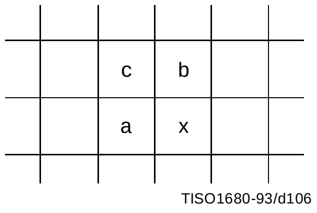


Figure H.1 – Relationship between sample and prediction samples

Define Px to be the prediction and Ra, Rb, and Rc to be the reconstructed samples immediately to the left, immediately above, and diagonally to the left of the current sample. The allowed predictors, one of which is selected in the scan header, are listed in Table H.1.

Table H.1 – Predictors for lossless coding

Selection-value	Prediction
0	No prediction (See Annex J)
1	Px = Ra
2	Px = Rb
3	Px = Rc
4	Px = Ra + Rb – Rc
5	Px = Ra + ((Rb – Rc)/2)[a)]
6	Px = Rb + ((Ra – Rc)/2)[a)]
7	Px = (Ra + Rb)/2
a) Shift right arithmetic operation	

Selection-value 0 shall only be used for differential coding in the hierarchical mode of operation. Selections 1, 2 and 3 are one-dimensional predictors and selections 4, 5, 6, and 7 are two-dimensional predictors.

The one-dimensional horizontal predictor (prediction sample Ra) is used for the first line of samples at the start of the scan and at the beginning of each restart interval. The selected predictor is used for all other lines. The sample from the line above (prediction sample Rb) is used at the start of each line, except for the first line. At the beginning of the first line and at the beginning of each restart interval the prediction value of 2^{P-1} is used, where P is the input precision.

If the point transformation parameter (see A.4) is non-zero, the prediction value at the beginning of the first lines and the beginning of each restart interval is $2^{P - Pt - 1}$, where Pt is the value of the point transformation parameter.

Each prediction is calculated with full integer arithmetic precision, and without clamping of either underflow or overflow beyond the input precision bounds. For example, if Ra and Rb are both 16-bit integers, the sum is a 17-bit integer. After dividing the sum by 2 (predictor 7), the prediction is a 16-bit integer.

For simplicity of implementation, the divide by 2 in the prediction selections 5 and 6 of Table H.1 is done by an arithmetic-right-shift of the integer values.

The difference between the prediction value and the input is calculated modulo 2^{16}. In the decoder the difference is decoded and added, modulo 2^{16}, to the prediction.

H.1.2.2 Huffman coding of the modulo difference

The Huffman coding procedures defined in Annex F for coding the DC coefficients are used to code the modulo 2^{16} differences. The table for DC coding contained in Tables F.1 and F.6 is extended by one additional entry. No extra bits are appended after SSSS = 16 is encoded. See Table H.2.

Table H.2 – Difference categories for lossless Huffman coding

SSSS	Difference values
0	0
1	–1,1
2	–3,–2,2,3
3	–7..–4,4..7
4	–15..–8,8..15
5	–31..–16,16..31
6	–63..–32,32..63
7	–127..–64,64..127
8	–255..–128,128..255
9	–511..–256,256..511
10	–1 023..–512,512..1 023
11	–2 047..–1 024,1 024..2 047
12	–4 095..–2 048,2 048..4 095
13	–8 191..–4 096,4 096..8 191
14	–16 383..–8 192,8 192..16 383
15	–32 767..–16 384,16 384..32 767
16	32 768

H.1.2.3 Arithmetic coding of the modulo difference

The statistical model defined for the DC coefficient arithmetic coding model (see F.1.4.4.1) is generalized to a two-dimensional form in which differences coded for the sample to the left and for the line above are used for conditioning.

H.1.2.3.1 Two-dimensional statistical model

The binary decisions are conditioned on the differences coded for the neighbouring samples immediately above and immediately to the left from the same component. As in the coding of the DC coefficients, the differences are classified into 5 categories: zero(0), small positive (+S), small negative (–S), large positive (+L), and large negative (–L). The two independent difference categories combine to give 25 different conditioning states. Figure H.2 shows the two-dimensional array of conditioning indices. For each of the 25 conditioning states probability estimates for four binary decisions are kept.

At the beginning of the scan and each restart interval the conditioning derived from the line above is set to zero for the first line of each component. At the start of each line, the difference to the left is set to zero for the purposes of calculating the conditioning.

Difference above (position b)

Difference to left (position a)	0	+S	–S	+L	–L
0	0	4	8	12	16
+S	20	24	28	32	36
–S	40	44	48	52	56
+L	60	64	68	72	76
–L	80	84	88	92	96

TISO1690-93/d107

Figure H.2 – 5 × 5 Conditioning array for two-dimensional statistical model

H.1.2.3.2 Assignment of statistical bins to the DC binary decision tree

Each statistics area for lossless coding consists of a contiguous set of 158 statistics bins. The first 100 bins consist of 25 sets of four bins selected by a context-index S0. The value of S0 is given by L_Context(Da,Db), which provides a value of 0, 4,..., 92 or 96, depending on the difference classifications of Da and Db (see H.1.2.3.1). The value for S0 provided by L_Context(Da,Db) is from the array in Figure H.2.

The remaining 58 bins consist of two sets of 29 bins, X1, ..., X15, M2, ..., M15, which are used to code magnitude category decisions and magnitude bits. The value of X1 is given by X1_Context(Db), which provides a value of 100 when Db is in the zero, small positive or small negative categories and a value of 129 when Db is in the large positive or large negative categories.

The assignment of statistical bins to the binary decision tree used for coding the difference is given in Table H.3.

Table H.3 – Statistical model for lossless coding

Context-index	Value	Coding decision
S0	L_Context(Da,Db)	V = 0
SS	S0 + 1	Sign
SP	S0 + 2	Sz < 1 if V > 0
SN	S0 + 3	Sz < 1 if V < 0
X1	X1_Context(Db)	Sz < 2
X2	X1 + 1	Sz < 4
X3	X1 + 2	Sz < 8
.	.	.
.	.	.
X15	X1 + 14	Sz < 2^{15}
M2	X2 + 14	Magnitude bits if Sz < 4
M3	X3 + 14	Magnitude bits if Sz < 8
.	.	.
.	.	.
M15	X15 + 14	Magnitude bits if Sz < 2^{15}

H.1.2.3.3 Default conditioning bounds

The bounds, L and U, for determining the conditioning category have the default values L = 0 and U = 1. Other bounds may be set using the DAC (Define-Arithmetic-Conditioning) marker segment, as described in Annex B.

H.1.2.3.4 Initial conditions for statistical model

At the start of a scan and at each restart, all statistics bins are re-initialized to the standard default value described in Annex D.

H.2 Lossless decoder processes

Lossless decoders may employ either Huffman decoding or arithmetic decoding. They shall be capable of using up to four tables in a scan. Lossless decoders shall be able to decode encoded image source data with any input precision from 2 to 16 bits per sample.

H.2.1 Lossless decoder control procedures

Subclause E.2 contains the decoder control procedures. In applying these procedures to the lossless decoder the data unit is one sample.

When the decoder is reset in the restart interval control procedure (see E.2.4) the prediction is reset to the same value used in the encoder (see H.1.2.1). If arithmetic coding is used, the statistics are also reset.

Restrictions on the restart interval are specified in H.1.1.

H.2.2 Coding model for lossless decoding

The predictor calculations defined in H.1.2 also apply to the lossless decoder processes.

The lossless decoders, decode the differences and add them, modulo 2^{16}, to the predictions to create the output. The lossless decoders shall be able to interpret the point transform parameter, and if non-zero, multiply the output of the lossless decoder by 2^{Pt}.

In order to avoid repetition, detailed flow charts of the lossless decoding procedures are omitted.

Annex J

Hierarchical mode of operation

(This annex forms an integral part of this Recommendation | International Standard)

This annex provides a **functional specification** of the coding processes for the hierarchical mode of operation.

In the hierarchical mode of operation each component is encoded or decoded in a non-differential frame. Such frames may be followed by a sequence of differential frames. A non-differential frame shall be encoded or decoded using the procedures defined in Annexes F, G and H. Differential frame procedures are defined in this annex.

The coding process for a hierarchical encoding containing DCT-based processes is defined as the highest numbered process listed in Table J.1 which is used to code any non-differential DCT-based or differential DCT-based frame in the compressed image data format. The coding process for a hierarchical encoding containing only lossless processes is defined to be the process used for the non-differential frames.

Table J.1 – Coding processes for hierarchical mode

Process	Non-differential frame specification	
1	Extended sequential DCT, Huffman, 8-bit	Annex F, process 2
2	Extended sequential DCT, arithmetic, 8-bit	Annex F, process 3
3	Extended sequential DCT, Huffman, 12-bit	Annex F, process 4
4	Extended sequential DCT, arithmetic, 12-bit	Annex F, process 5
5	Spectral selection only, Huffman, 8-bit	Annex G, process 1
6	Spectral selection only, arithmetic, 8-bit	Annex G, process 2
7	Full progression, Huffman, 8-bit	Annex G, process 3
8	Full progression, arithmetic, 8-bit	Annex G, process 4
9	Spectral selection only, Huffman, 12-bit	Annex G, process 5
10	Spectral selection only, arithmetic, 12-bit	Annex G, process 6
11	Full progression, Huffman, 12-bit	Annex G, process 7
12	Full progression, arithmetic, 12-bit	Annex G, process 8
13	Lossless, Huffman, 2 through 16 bits	Annex H, process 1
14	Lossless, arithmetic, 2 through 16 bits	Annex H, process 2

Hierarchical mode syntax requires a DHP marker segment that appears before the non-differential frame or frames. It may include EXP marker segments and differential frames which shall follow the initial non-differential frame. The frame structure in hierarchical mode is identical to the frame structure in non-hierarchical mode.

Either all non-differential frames within an image shall be coded with DCT-based processes, or all non-differential frames shall be coded with lossless processes. All frames within an image must use the same entropy coding procedure, either Huffman or arithmetic, with the exception that non-differential frames coded with the baseline process may occur in the same image with frames coded with arithmetic coding processes.

If the non-differential frames use DCT-based processes, all differential frames except the final frame for a component shall use DCT-based processes. The final differential frame for each component may use a differential lossless process.

If the non-differential frames use lossless processes, all differential frames shall use differential lossless processes.

For each of the processes listed in Table J.1, the encoding processes are specified in J.1, and decoding processes are specified in J.2.

NOTE – There is **no requirement** in this Specification that any encoder or decoder which embodies one of the above-named processes shall implement the procedures in precisely the manner specified by the flow charts in this annex. It is necessary only that an encoder or decoder implement the **function** specified in this annex. The sole criterion for an encoder or decoder to be considered in compliance with this Specification is that it satisfy the requirements given in clause 6 (for encoders) or clause 7 (for decoders), as determined by the compliance tests specified in Part 2.

In the hierarchical mode of operation each component is encoded or decoded in a non-differential frame followed by a sequence of differential frames. A non-differential frame shall use the procedures defined in Annexes F, G, and H. Differential frame procedures are defined in this annex.

J.1 Hierarchical encoding

J.1.1 Hierarchical control procedure for encoding an image

The control structure for encoding of an image using the hierarchical mode is given in Figure J.1.

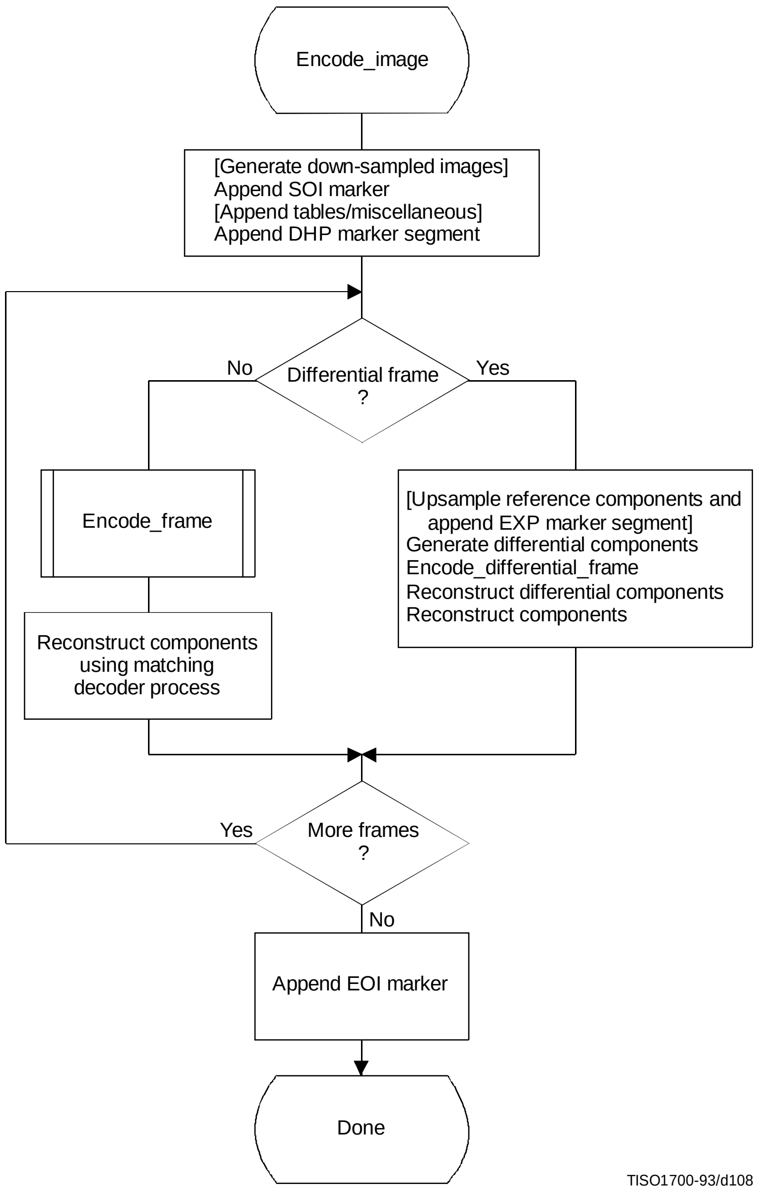

Encode_image
[Generate down-sampled images]
Append SOI marker
[Append tables/miscellaneous]
Append DHP marker segment
No
Differential frame ?
Yes
Encode_frame
[Upsample reference components and append EXP marker segment]
Generate differential components
Encode_differential_frame
Reconstruct differential components
Reconstruct components
Reconstruct components using matching decoder process
Yes
More frames ?
No
Append EOI marker
Done
TISO1700-93/d108

Figure J.1 – Hierarchical control procedure for encoding an image

In Figure J.1 procedures in brackets shall be performed whenever the particular hierarchical encoding sequence being followed requires them.

In the hierarchical mode the define-hierarchical-progression (DHP) marker segment shall be placed in the compressed image data before the first start-of-frame. The DHP segment is used to signal the size of the image components of the completed image. The syntax of the DHP segment is specified in Annex B.

The first frame for each component or group of components in a hierarchical process shall be encoded by a non-differential frame. Differential frames shall then be used to encode the two's complement differences between source input components (possibly downsampled) and the reference components (possibly upsampled). The reference components are reconstructed components created by previous frames in the hierarchical process. For either differential or non-differential frames, reconstructions of the components shall be generated if needed as reference components for a subsequent frame in the hierarchical process.

Resolution changes may occur between hierarchical frames in a hierarchical process. These changes occur if downsampling filters are used to reduce the spatial resolution of some or all of the components of the source image. When the resolution of a reference component does not match the resolution of the component input to a differential frame, an upsampling filter shall be used to increase the spatial resolution of the reference component. The EXP marker segment shall be added to the compressed image data before the start-of-frame whenever upsampling of a reference component is required. No more than one EXP marker segment shall precede a given frame.

Any of the marker segments allowed before a start-of-frame for the encoding process selected may be used before either non-differential or differential frames.

For 16-bit input precision (lossless encoder), the differential components which are input to a differential frame are calculated modulo 2^{16}. The reconstructed components calculated from the reconstructed differential components are also calculated modulo 2^{16}.

If a hierarchical encoding process uses a DCT encoding process for the first frame, all frames in the hierarchical process except for the final frame for each component shall use the DCT encoding processes defined in either Annex F or Annex G, or the modified DCT encoding processes defined in this annex. The final frame may use a modified lossless process defined in this annex.

If a hierarchical encoding process uses a lossless encoding process for the first frame, all frames in the hierarchical process shall use a lossless encoding process defined in Annex H, or a modified lossless process defined in this annex.

J.1.1.1 Downsampling filter

The downsampled components are generated using a downsampling filter that is not specified in this Specification. This filter should, however, be consistent with the upsampling filter. An example of a downsampling filter is provided in K.5.

J.1.1.2 Upsampling filter

The upsampling filter increases the spatial resolution by a factor of two horizontally, vertically, or both. Bi-linear interpolation is used for the upsampling filter, as illustrated in Figure J.2.

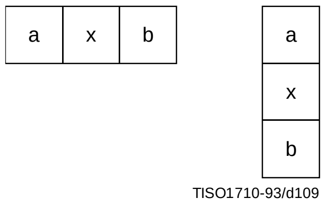


Figure J.2 – Diagram of sample positions for upsampling rules

The rule for calculating the interpolated value is:

$$P_x = (Ra + Rb) / 2$$

where Ra and Rb are sample values from adjacent positions a and b of the lower resolution image and Px is the interpolated value. The division indicates truncation, not rounding. The left-most column of the upsampled image matches the left-most column of the lower resolution image. The top line of the upsampled image matches the top line of the lower resolution image. The right column and the bottom line of the lower resolution image are replicated to provide the values required for the right column edge and bottom line interpolations. The upsampling process always doubles the line length or the number of lines.

If both horizontal and vertical expansions are signalled, they are done in sequence – first the horizontal expansion and then the vertical.

J.1.2 Control procedure for encoding a differential frame

The control procedures in Annex E for frames, scans, restart intervals, and MCU also apply to the encoding of differential frames, and the scans, restart intervals, and MCU from which the differential frame is constructed. The differential frames differ from the frames of Annexes F, G, and H only at the coding model level.

J.1.3 Encoder coding models for differential frames

The coding models defined in Annexes F, G, and H are modified to allow them to be used for coding of two's complement differences.

J.1.3.1 Modifications to encoder DCT encoding models for differential frames

Two modifications are made to the DCT coding models to allow them to be used in differential frames. First, the FDCT of the differential input is calculated without the level shift. Second, the DC coefficient of the DCT is coded directly – without prediction.

J.1.3.2 Modifications to lossless encoding models for differential frames

One modification is made to the lossless coding models. The difference is coded directly – without prediction. The prediction selection parameter in the scan header shall be set to zero. The point transform which may be applied to the differential inputs is defined in Annex A.

J.1.4 Modifications to the entropy encoders for differential frames

The coding of two's complement differences requires one extra bit of precision for the Huffman coding of AC coefficients. The extension to Tables F.1 and F.7 is given in Table J.2.

Table J.2 – Modifications to table of AC coefficient amplitude ranges

SSSS	AC coefficients
15	–32 767..–16 384, 16 384..32 767

The arithmetic coding models are already defined for the precision needed in differential frames.

J.2 Hierarchical decoding

J.2.1 Hierarchical control procedure for decoding an image

The control structure for decoding an image using the hierarchical mode is given in Figure J.3.

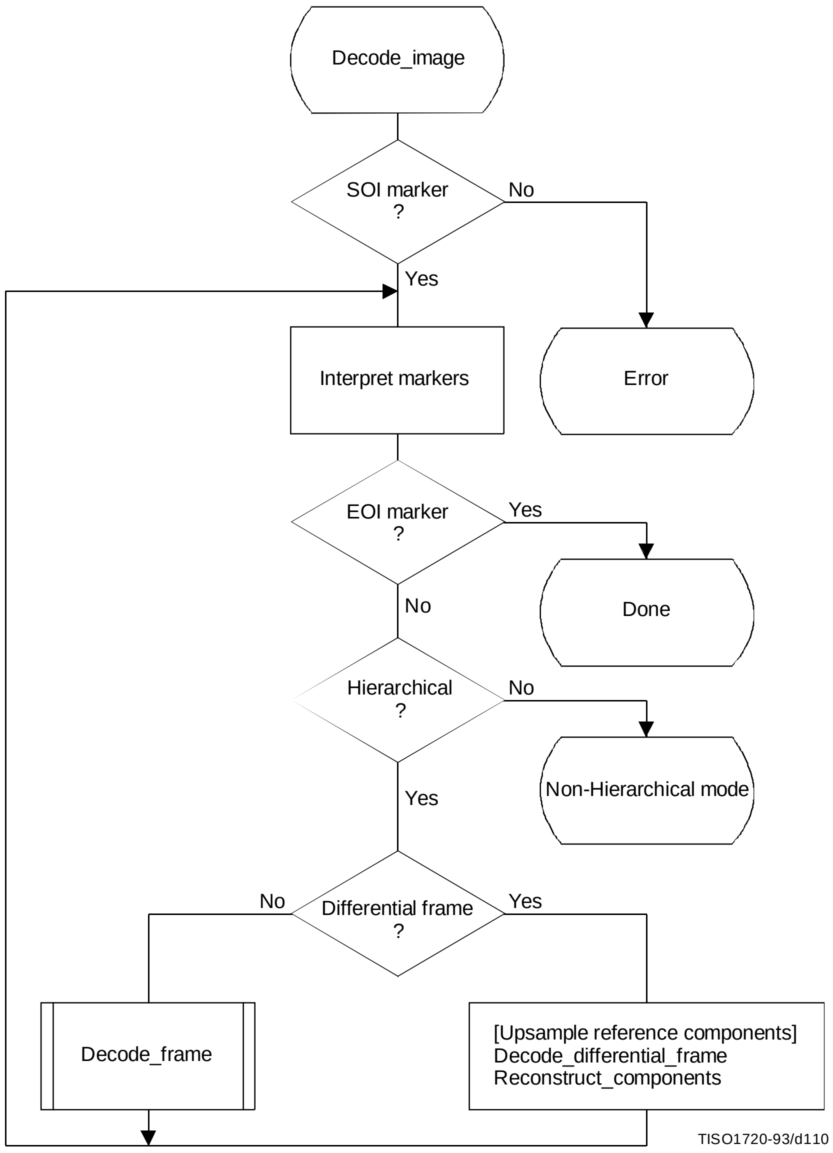


Figure J.3 – Hierarchical control procedure for decoding an image

The Interpret markers procedure shall decode the markers which may precede the SOF marker, continuing this decoding until either a SOF or EOI marker is found. If the DHP marker is encountered before the first frame, a flag is set which selects the hierarchical decoder at the "hierarchical?" decision point. In addition to the DHP marker (which shall precede any SOF) and the EXP marker (which shall precede any differential SOF requiring resolution changes in the reference components), any other markers which may precede a SOF shall be interpreted to the extent required for decoding of the compressed image data.

If a differential SOF marker is found, the differential frame path is followed. If the EXP was encountered in the Interpret markers procedure, the reference components for the frame shall be upsampled as required by the parameters in the EXP segment. The upsampling procedure described in J.1.1.2 shall be followed.

The Decode_differential_frame procedure generates a set of differential components. These differential components shall be added, modulo 2^{16}, to the upsampled reference components in the Reconstruct_components procedure. This creates a new set of reference components which shall be used when required in subsequent frames of the hierarchical process.

J.2.2 Control procedure for decoding a differential frame

The control procedures in Annex E for frames, scans, restart intervals, and MCU also apply to the decoding of differential frames and the scans, restart intervals, and MCU from which the differential frame is constructed. The differential frame differs from the frames of Annexes F, G, and H only at the decoder coding model level.

J.2.3 Decoder coding models for differential frames

The decoding models described in Annexes F, G, and H are modified to allow them to be used for decoding of two's complement differential components.

J.2.3.1 Modifications to the differential frame decoder DCT coding model

Two modifications are made to the decoder DCT coding models to allow them to code differential frames. First, the IDCT of the differential output is calculated without the level shift. Second, the DC coefficient of the DCT is decoded directly – without prediction.

J.2.3.2 Modifications to the differential frame decoder lossless coding model

One modification is made to the lossless decoder coding model. The difference is decoded directly – without prediction. If the point transformation parameter in the scan header is not zero, the point transform, defined in Annex A, shall be applied to the differential output.

J.2.4 Modifications to the entropy decoders for differential frames

The decoding of two's complement differences requires one extra bit of precision in the Huffman code table. This is described in J.1.4. The arithmetic coding models are already defined for the precision needed in differential frames.

Annex K

Examples and guidelines

(This annex does not form an integral part of this Recommendation | International Standard)

This annex provides examples of various tables, procedures, and other guidelines.

K.1 Quantization tables for luminance and chrominance components

Two examples of quantization tables are given in Tables K.1 and K.2. These are based on psychovisual thresholding and are derived empirically using luminance and chrominance and 2:1 horizontal subsampling. These tables are provided as examples only and are not necessarily suitable for any particular application. These quantization values have been used with good results on 8-bit per sample luminance and chrominance images of the format illustrated in Figure 13. Note that these quantization values are appropriate for the DCT normalization defined in A.3.3.

If these quantization values are divided by 2, the resulting reconstructed image is usually nearly indistinguishable from the source image.

Table K.1 – Luminance quantization table

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Table K.2 – Chrominance quantization table

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

K.2 A procedure for generating the lists which specify a Huffman code table

A Huffman table is generated from a collection of statistics in two steps. The first step is the generation of the list of lengths and values which are in accord with the rules for generating the Huffman code tables. The second step is the generation of the Huffman code table from the list of lengths and values.

The first step, the topic of this section, is needed only for custom Huffman table generation and is done only in the encoder. In this step the statistics are used to create a table associating each value to be coded with the size (in bits) of the corresponding Huffman code. This table is sorted by code size.

A procedure for creating a Huffman table for a set of up to 256 symbols is shown in Figure K.1. Three vectors are defined for this procedure:

FREQ(V)	Frequency of occurrence of symbol V
CODESIZE(V)	Code size of symbol V
OTHERS(V)	Index to next symbol in chain of all symbols in current branch of code tree

where V goes from 0 to 256.

Before starting the procedure, the values of FREQ are collected for V = 0 to 255 and the FREQ value for V = 256 is set to 1 to reserve one code point. FREQ values for unused symbols are defined to be zero. In addition, the entries in CODESIZE are all set to 0, and the indices in OTHERS are set to –1, the value which terminates a chain of indices. Reserving one code point guarantees that no code word can ever be all "1" bits.

The search for the entry with the least value of FREQ(V) selects the largest value of V with the least value of FREQ(V) greater than zero.

The procedure "Find V1 for least value of FREQ(V1) > 0" always selects the value with the largest value of V1 when more than one V1 with the same frequency occurs. The reserved code point is then guaranteed to be in the longest code word category.

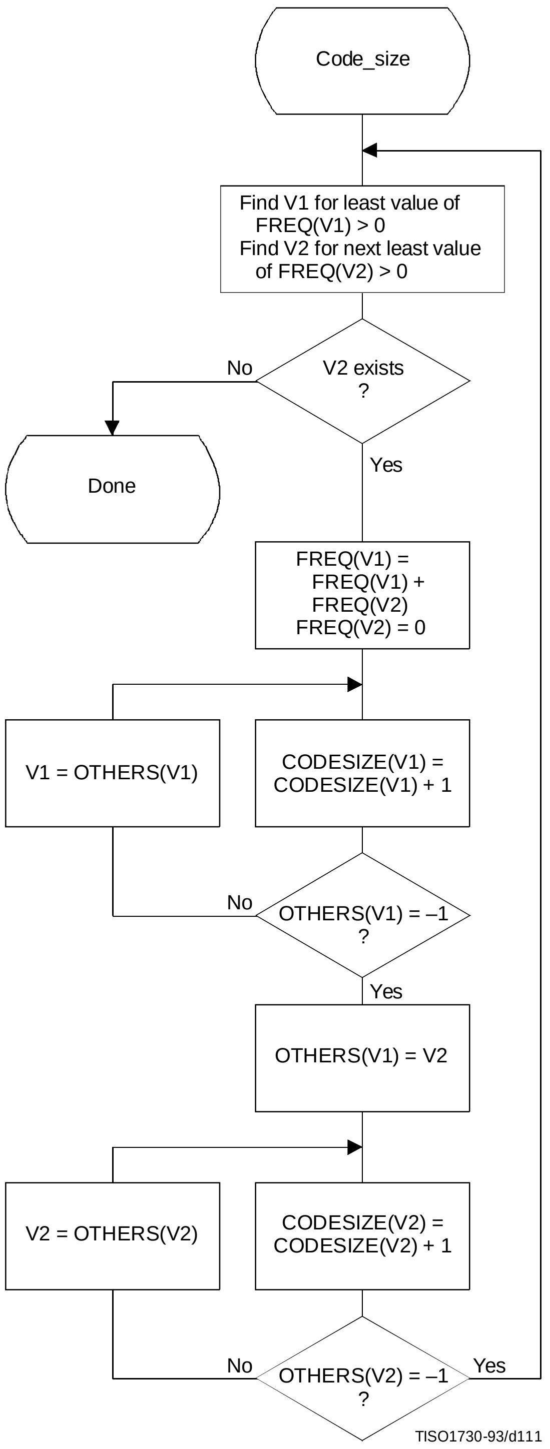

Code_size
Find V1 for least value of FREQ(V1) > 0
Find V2 for next least value of FREQ(V2) > 0
V2 exists ?
No
Yes
Done
FREQ(V1) = FREQ(V1) + FREQ(V2)
FREQ(V2) = 0
V1 = OTHERS(V1)
CODESIZE(V1) = CODESIZE(V1) + 1
OTHERS(V1) = –1 ?
No
Yes
OTHERS(V1) = V2
V2 = OTHERS(V2)
CODESIZE(V2) = CODESIZE(V2) + 1
OTHERS(V2) = –1 ?
No
Yes
TISO1730-93/d111

Figure K.1 – Procedure to find Huffman code sizes

Once the code lengths for each symbol have been obtained, the number of codes of each length is obtained using the procedure in Figure K.2. The count for each size is contained in the list, BITS. The counts in BITS are zero at the start of the procedure. The procedure assumes that the probabilities are large enough that code lengths greater than 32 bits never occur. Note that until the final Adjust_BITS procedure is complete, BITS may have more than the 16 entries required in the table specification (see Annex C).

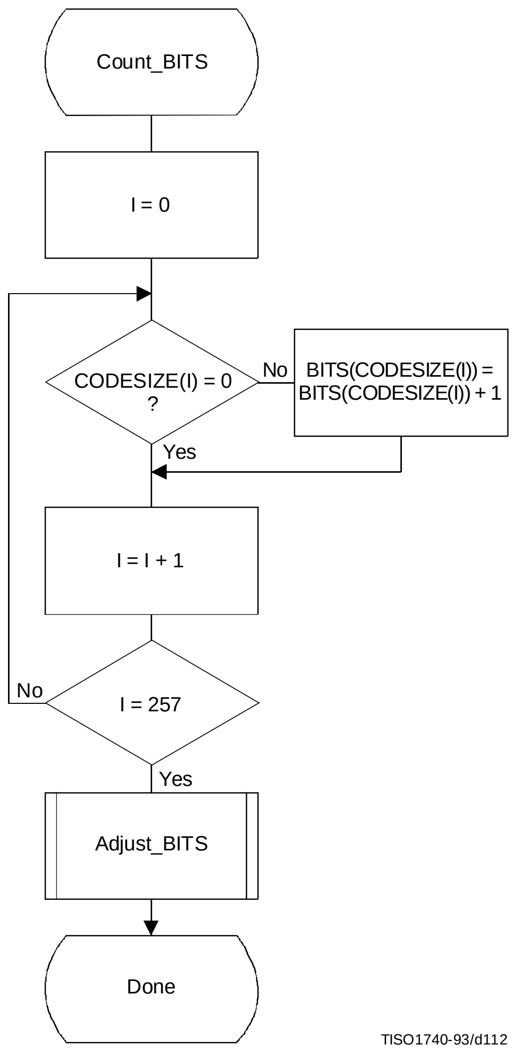


Figure K.2 – Procedure to find the number of codes of each size

Figure K.3 gives the procedure for adjusting the BITS list so that no code is longer than 16 bits. Since symbols are paired for the longest Huffman code, the symbols are removed from this length category two at a time. The prefix for the pair (which is one bit shorter) is allocated to one of the pair; then (skipping the BITS entry for that prefix length) a code word from the next shortest non-zero BITS entry is converted into a prefix for two code words one bit longer. After the BITS list is reduced to a maximum code length of 16 bits, the last step removes the reserved code point from the code length count.

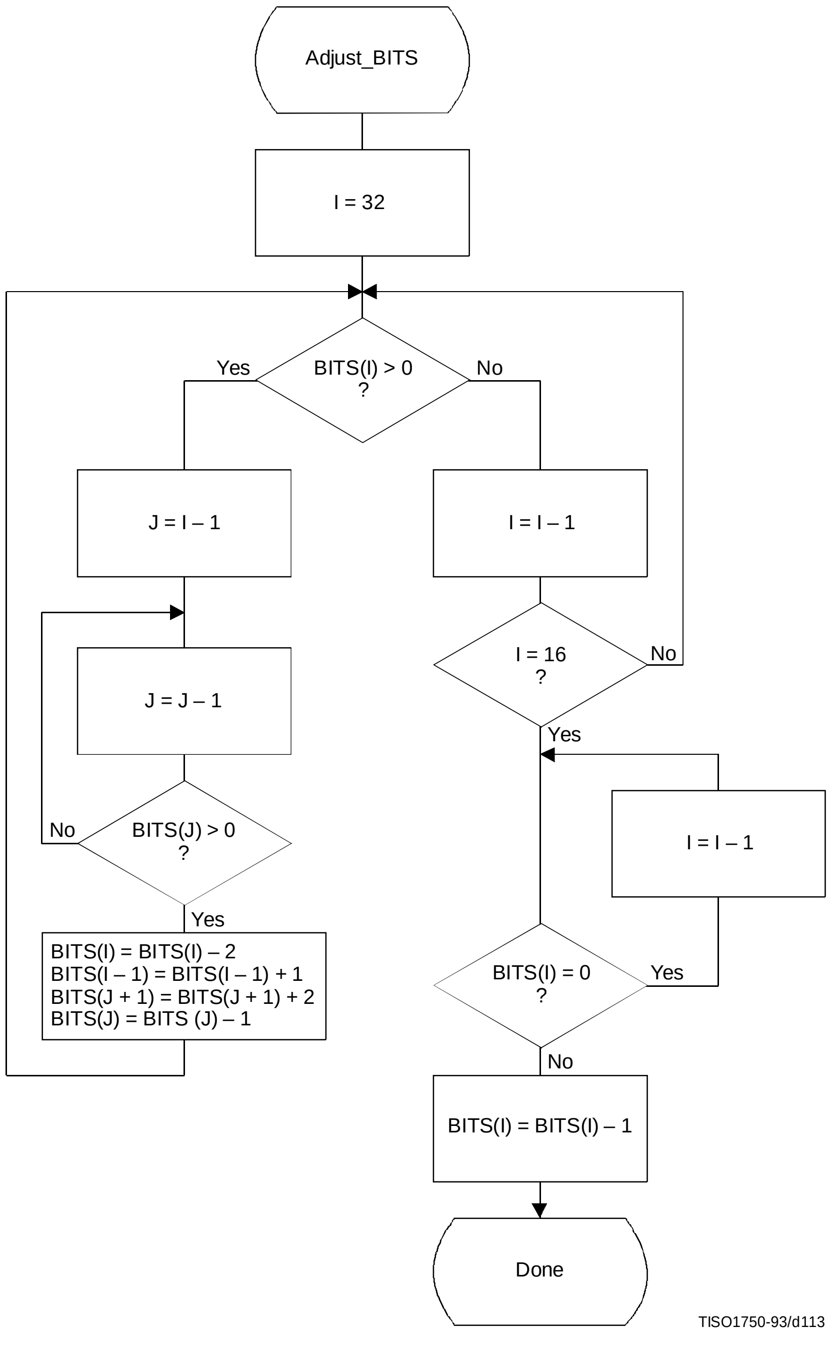


Figure K.3 – Procedure for limiting code lengths to 16 bits

The input values are sorted according to code size as shown in Figure K.4. HUFFVAL is the list containing the input values associated with each code word, in order of increasing code length.

At this point, the list of code lengths (BITS) and the list of values (HUFFVAL) can be used to generate the code tables. These procedures are described in Annex C.

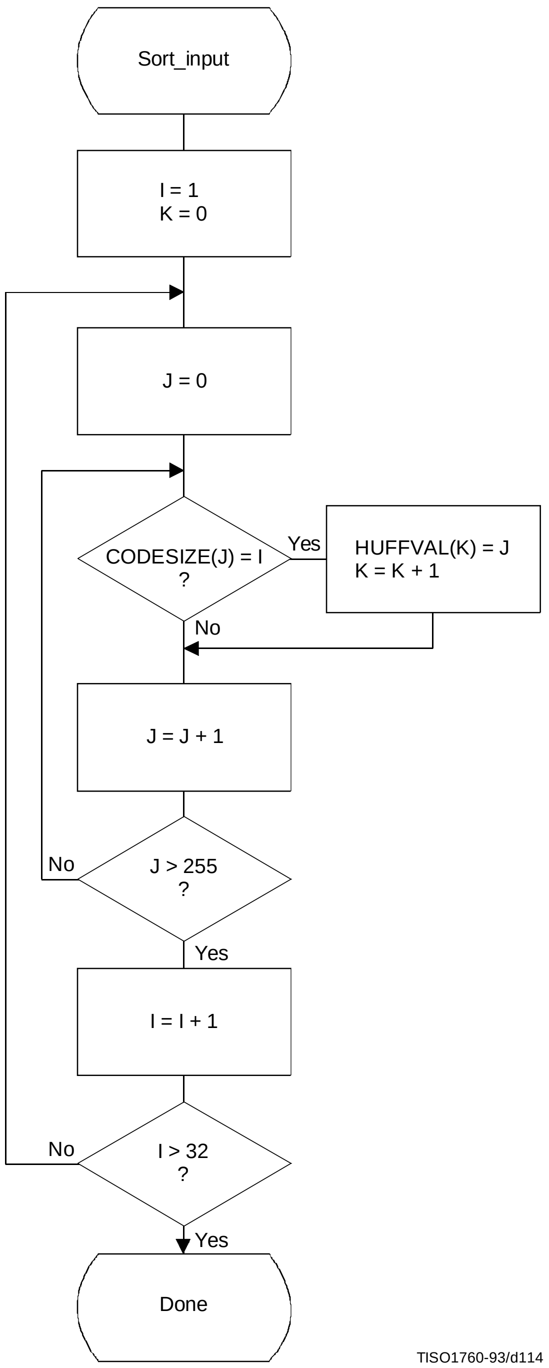


Figure K.4 – Sorting of input values according to code size

K.3 Typical Huffman tables for 8-bit precision luminance and chrominance

Huffman table-specification syntax is specified in B.2.4.2.

K.3.1 Typical Huffman tables for the DC coefficient differences

Tables K.3 and K.4 give Huffman tables for the DC coefficient differences which have been developed from the average statistics of a large set of video images with 8-bit precision. Table K.3 is appropriate for luminance components and Table K.4 is appropriate for chrominance components. Although there are no default tables, these tables may prove to be useful for many applications.

Table K.3 – Table for luminance DC coefficient differences

Category	Code length	Code word
0	2	00
1	3	010
2	3	011
3	3	100
4	3	101
5	3	110
6	4	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110

Table K.4 – Table for chrominance DC coefficient differences

Category	Code length	Code word
0	2	00
1	2	01
2	2	10
3	3	110
4	4	1110
5	5	11110
6	6	111110
7	7	1111110
8	8	11111110
9	9	111111110
10	10	1111111110
11	11	11111111110

K.3.2 Typical Huffman tables for the AC coefficients

Tables K.5 and K.6 give Huffman tables for the AC coefficients which have been developed from the average statistics of a large set of images with 8-bit precision. Table K.5 is appropriate for luminance components and Table K.6 is appropriate for chrominance components. Although there are no default tables, these tables may prove to be useful for many applications.

Table K.5 – Table for luminance AC coefficients (sheet 1 of 4)

Run/Size	Code length	Code word
0/0 (EOB)	4	1010
0/1	2	00
0/2	2	01
0/3	3	100
0/4	4	1011
0/5	5	11010
0/6	7	1111000
0/7	8	11111000
0/8	10	1111110110
0/9	16	1111111110000010
0/A	16	1111111110000011
1/1	4	1100
1/2	5	11011
1/3	7	1111001
1/4	9	111110110
1/5	11	11111110110
1/6	16	1111111110000100
1/7	16	1111111110000101
1/8	16	1111111110000110
1/9	16	1111111110000111
1/A	16	1111111110001000
2/1	5	11100
2/2	8	11111001
2/3	10	1111110111
2/4	12	111111110100
2/5	16	1111111110001001
2/6	16	1111111110001010
2/7	16	1111111110001011
2/8	16	1111111110001100
2/9	16	1111111110001101
2/A	16	1111111110001110
3/1	6	111010
3/2	9	111110111
3/3	12	111111110101
3/4	16	1111111110001111
3/5	16	1111111110010000
3/6	16	1111111110010001
3/7	16	1111111110010010
3/8	16	1111111110010011
3/9	16	1111111110010100
3/A	16	1111111110010101

Table K.5 (sheet 2 of 4)

Run/Size	Code length	Code word
4/1	6	111011
4/2	10	1111111000
4/3	16	1111111110010110
4/4	16	1111111110010111
4/5	16	1111111110011000
4/6	16	1111111110011001
4/7	16	1111111110011010
4/8	16	1111111110011011
4/9	16	1111111110011100
4/A	16	1111111110011101
5/1	7	1111010
5/2	11	11111110111
5/3	16	1111111110011110
5/4	16	1111111110011111
5/5	16	1111111110100000
5/6	16	1111111110100001
5/7	16	1111111110100010
5/8	16	1111111110100011
5/9	16	1111111110100100
5/A	16	1111111110100101
6/1	7	1111011
6/2	12	111111110110
6/3	16	1111111110100110
6/4	16	1111111110100111
6/5	16	1111111110101000
6/6	16	1111111110101001
6/7	16	1111111110101010
6/8	16	1111111110101011
6/9	16	1111111110101100
6/A	16	1111111110101101
7/1	8	11111010
7/2	12	111111110111
7/3	16	1111111110101110
7/4	16	1111111110101111
7/5	16	1111111110110000
7/6	16	1111111110110001
7/7	16	1111111110110010
7/8	16	1111111110110011
7/9	16	1111111110110100
7/A	16	1111111110110101
8/1	9	111111000
8/2	15	111111111000000

Table K.5 (sheet 3 of 4)

Run/Size	Code length	Code word
8/3	16	1111111110110110
8/4	16	1111111110110111
8/5	16	1111111110111000
8/6	16	1111111110111001
8/7	16	1111111110111010
8/8	16	1111111110111011
8/9	16	1111111110111100
8/A	16	1111111110111101
9/1	9	111111001
9/2	16	1111111110111110
9/3	16	1111111110111111
9/4	16	1111111111000000
9/5	16	1111111111000001
9/6	16	1111111111000010
9/7	16	1111111111000011
9/8	16	1111111111000100
9/9	16	1111111111000101
9/A	16	1111111111000110
A/1	9	111111010
A/2	16	1111111111000111
A/3	16	1111111111001000
A/4	16	1111111111001001
A/5	16	1111111111001010
A/6	16	1111111111001011
A/7	16	1111111111001100
A/8	16	1111111111001101
A/9	16	1111111111001110
A/A	16	1111111111001111
B/1	10	1111111001
B/2	16	1111111111010000
B/3	16	1111111111010001
B/4	16	1111111111010010
B/5	16	1111111111010011
B/6	16	1111111111010100
B/7	16	1111111111010101
B/8	16	1111111111010110
B/9	16	1111111111010111
B/A	16	1111111111011000
C/1	10	1111111010
C/2	16	1111111111011001
C/3	16	1111111111011010
C/4	16	1111111111011011

Table K.5 (sheet 4 of 4)

Run/Size	Code length	Code word
C/5	16	1111111111011100
C/6	16	1111111111011101
C/7	16	1111111111011110
C/8	16	1111111111011111
C/9	16	1111111111100000
C/A	16	1111111111100001
D/1	11	11111111000
D/2	16	1111111111100010
D/3	16	1111111111100011
D/4	16	1111111111100100
D/5	16	1111111111100101
D/6	16	1111111111100110
D/7	16	1111111111100111
D/8	16	1111111111101000
D/9	16	1111111111101001
D/A	16	1111111111101010
E/1	16	1111111111101011
E/2	16	1111111111101100
E/3	16	1111111111101101
E/4	16	1111111111101110
E/5	16	1111111111101111
E/6	16	1111111111110000
E/7	16	1111111111110001
E/8	16	1111111111110010
E/9	16	1111111111110011
E/A	16	1111111111110100
F/0 (ZRL)	11	11111111001
F/1	16	1111111111110101
F/2	16	1111111111110110
F/3	16	1111111111110111
F/4	16	1111111111111000
F/5	16	1111111111111001
F/6	16	1111111111111010
F/7	16	1111111111111011
F/8	16	1111111111111100
F/9	16	1111111111111101
F/A	16	1111111111111110

Table K.6 – Table for chrominance AC coefficients (sheet 1 of 4)

Run/Size	Code length	Code word
0/0 (EOB)	2	00
0/1	2	01
0/2	3	100
0/3	4	1010
0/4	5	11000
0/5	5	11001
0/6	6	111000
0/7	7	1111000
0/8	9	111110100
0/9	10	1111110110
0/A	12	111111110100
1/1	4	1011
1/2	6	111001
1/3	8	11110110
1/4	9	111110101
1/5	11	11111110110
1/6	12	111111110101
1/7	16	1111111110001000
1/8	16	1111111110001001
1/9	16	1111111110001010
1/A	16	1111111110001011
2/1	5	11010
2/2	8	11110111
2/3	10	1111110111
2/4	12	111111110110
2/5	15	111111111000010
2/6	16	1111111110001100
2/7	16	1111111110001101
2/8	16	1111111110001110
2/9	16	1111111110001111
2/A	16	1111111110010000
3/1	5	11011
3/2	8	11111000
3/3	10	1111111000
3/4	12	111111110111
3/5	16	1111111110010001
3/6	16	1111111110010010
3/7	16	1111111110010011
3/8	16	1111111110010100
3/9	16	1111111110010101
3/A	16	1111111110010110
4/1	6	111010

Table K.6 (sheet 2 of 4)

Run/Size	Code length	Code word
4/2	9	111110110
4/3	16	1111111110010111
4/4	16	1111111110011000
4/5	16	1111111110011001
4/6	16	1111111110011010
4/7	16	1111111110011011
4/8	16	1111111110011100
4/9	16	1111111110011101
4/A	16	1111111110011110
5/1	6	111011
5/2	10	1111111001
5/3	16	1111111110011111
5/4	16	1111111110100000
5/5	16	1111111110100001
5/6	16	1111111110100010
5/7	16	1111111110100011
5/8	16	1111111110100100
5/9	16	1111111110100101
5/A	16	1111111110100110
6/1	7	1111001
6/2	11	11111110111
6/3	16	1111111110100111
6/4	16	1111111110101000
6/5	16	1111111110101001
6/6	16	1111111110101010
6/7	16	1111111110101011
6/8	16	1111111110101100
6/9	16	1111111110101101
6/A	16	1111111110101110
7/1	7	1111010
7/2	11	11111111000
7/3	16	1111111110101111
7/4	16	1111111110110000
7/5	16	1111111110110001
7/6	16	1111111110110010
7/7	16	1111111110110011
7/8	16	1111111110110100
7/9	16	1111111110110101
7/A	16	1111111110110110
8/1	8	11111001
8/2	16	1111111110110111
8/3	16	1111111110111000

Table K.6 (sheet 3 of 4)

Run/Size	Code length	Code word
8/4	16	1111111110111001
8/5	16	1111111110111010
8/6	16	1111111110111011
8/7	16	1111111110111100
8/8	16	1111111110111101
8/9	16	1111111110111110
8/A	16	1111111110111111
9/1	9	111110111
9/2	16	1111111111000000
9/3	16	1111111111000001
9/4	16	1111111111000010
9/5	16	1111111111000011
9/6	16	1111111111000100
9/7	16	1111111111000101
9/8	16	1111111111000110
9/9	16	1111111111000111
9/A	16	1111111111001000
A/1	9	111111000
A/2	16	1111111111001001
A/3	16	1111111111001010
A/4	16	1111111111001011
A/5	16	1111111111001100
A/6	16	1111111111001101
A/7	16	1111111111001110
A/8	16	1111111111001111
A/9	16	1111111111010000
A/A	16	1111111111010001
B/1	9	111111001
B/2	16	1111111111010010
B/3	16	1111111111010011
B/4	16	1111111111010100
B/5	16	1111111111010101
B/6	16	1111111111010110
B/7	16	1111111111010111
B/8	16	1111111111011000
B/9	16	1111111111011001
B/A	16	1111111111011010
C/1	9	111111010
C/2	16	1111111111011011
C/3	16	1111111111011100
C/4	16	1111111111011101
C/5	16	1111111111011110

Table K.6 (sheet 4 of 4)

Run/Size	Code length	Code word
C/6	16	1111111111011111
C/7	16	1111111111100000
C/8	16	1111111111100001
C/9	16	1111111111100010
C/A	16	1111111111100011
D/1	11	11111111001
D/2	16	1111111111100100
D/3	16	1111111111100101
D/4	16	1111111111100110
D/5	16	1111111111100111
D/6	16	1111111111101000
D/7	16	1111111111101001
D/8	16	1111111111101010
D/9	16	1111111111101011
D/A	16	1111111111101100
E/1	14	11111111100000
E/2	16	1111111111101101
E/3	16	1111111111101110
E/4	16	1111111111101111
E/5	16	1111111111110000
E/6	16	1111111111110001
E/7	16	1111111111110010
E/8	16	1111111111110011
E/9	16	1111111111110100
E/A	16	1111111111110101
F/0 (ZRL)	10	1111111010
F/1	15	111111111000011
F/2	16	1111111111110110
F/3	16	1111111111110111
F/4	16	1111111111111000
F/5	16	1111111111111001
F/6	16	1111111111111010
F/7	16	1111111111111011
F/8	16	1111111111111100
F/9	16	1111111111111101
F/A	16	1111111111111110

K.3.3 Huffman table-specification examples

K.3.3.1 Specification of typical tables for DC difference coding

A set of typical tables for DC component coding is given in K.3.1. The specification of these tables is as follows:

For Table K.3 (for luminance DC coefficients), the 16 bytes which specify the list of code lengths for the table are

X’00 01 05 01 01 01 01 01 01 00 00 00 00 00 00 00’

The set of values following this list is

X’00 01 02 03 04 05 06 07 08 09 0A 0B’

For Table K.4 (for chrominance DC coefficients), the 16 bytes which specify the list of code lengths for the table are

X’00 03 01 01 01 01 01 01 01 01 01 00 00 00 00 00’

The set of values following this list is

X’00 01 02 03 04 05 06 07 08 09 0A 0B’

K.3.3.2 Specification of typical tables for AC coefficient coding

A set of typical tables for AC component coding is given in K.3.2. The specification of these tables is as follows:

For Table K.5 (for luminance AC coefficients), the 16 bytes which specify the list of code lengths for the table are

X’00 02 01 03 03 02 04 03 05 05 04 04 00 00 01 7D’

The set of values which follows this list is

```
X’01  02  03  00  04  11  05  12  21  31  41  06  13  51  61  07
  22  71  14  32  81  91  A1  08  23  42  B1  C1  15  52  D1  F0
  24  33  62  72  82  09  0A  16  17  18  19  1A  25  26  27  28
  29  2A  34  35  36  37  38  39  3A  43  44  45  46  47  48  49
  4A  53  54  55  56  57  58  59  5A  63  64  65  66  67  68  69
  6A  73  74  75  76  77  78  79  7A  83  84  85  86  87  88  89
  8A  92  93  94  95  96  97  98  99  9A  A2  A3  A4  A5  A6  A7
  A8  A9  AA  B2  B3  B4  B5  B6  B7  B8  B9  BA  C2  C3  C4  C5
  C6  C7  C8  C9  CA  D2  D3  D4  D5  D6  D7  D8  D9  DA  E1  E2
  E3  E4  E5  E6  E7  E8  E9  EA  F1  F2  F3  F4  F5  F6  F7  F8
  F9  FA’
```

For Table K.6 (for chrominance AC coefficients), the 16 bytes which specify the list of code lengths for the table are

X'00 02 01 02 04 04 03 04 07 05 04 04 00 01 02 77'

The set of values which follows this list is:

X'00	01	02	03	11	04	05	21	31	06	12	41	51	07	61	71
13	22	32	81	08	14	42	91	A1	B1	C1	09	23	33	52	F0
15	62	72	D1	0A	16	24	34	E1	25	F1	17	18	19	1A	26
27	28	29	2A	35	36	37	38	39	3A	43	44	45	46	47	48
49	4A	53	54	55	56	57	58	59	5A	63	64	65	66	67	68
69	6A	73	74	75	76	77	78	79	7A	82	83	84	85	86	87
88	89	8A	92	93	94	95	96	97	98	99	9A	A2	A3	A4	A5
A6	A7	A8	A9	AA	B2	B3	B4	B5	B6	B7	B8	B9	BA	C2	C3
C4	C5	C6	C7	C8	C9	CA	D2	D3	D4	D5	D6	D7	D8	D9	DA
E2	E3	E4	E5	E6	E7	E8	E9	EA	F2	F3	F4	F5	F6	F7	F8
F9	FA'														

K.4 Additional information on arithmetic coding

K.4.1 Test sequence for a small data set for the arithmetic coder

The following 256-bit test sequence (in hexadecimal form) is structured to test many of the encoder and decoder paths:

X'00020051 000000C0 0352872A AAAAAAAA 82C02000 FCD79EF6 74EAABF7 697EE74C'

Tables K.7 and K.8 provide a symbol-by-symbol list of the arithmetic encoder and decoder operation. In these tables the event count, EC, is listed first, followed by the value of Qe used in encoding and decoding that event. The decision D to be encoded (and decoded) is listed next. The column labeled MPS contains the sense of the MPS, and if it is followed by a CE (in the "CX" column), the conditional MPS/LPS exchange occurs when encoding and decoding the decision (see Figures D.3, D.4 and D.17). The contents of the A and C registers are the values before the event is encoded and decoded. ST is the number of X'FF' bytes stacked in the encoder waiting for a resolution of the carry-over. Note that the A register is always greater than X'7FFF'. (The starting value has an implied value of X'10000'.)

In the encoder test, the code bytes (B) are listed if they were completed during the coding of the preceding event. If additional bytes follow, they were also completed during the coding of the preceding event. If a byte is listed in the Bx column, the preceding byte in column B was modified by a carry-over.

In the decoder the code bytes are listed if they were placed in the code register just prior to the event EC.

For this file the coded bit count is 240, including the overhead to flush the final data from the C register. When the marker X'FFD9' is appended, a total of 256 bits are output. The actual compressed data sequence for the encoder is (in hexadecimal form)

X'655B5144 F7969D51 7855BFFF 00FC5184 C7CEF939 00287D46 708ECBC0 F6FFD900'

Table K.7 – Encoder test sequence (sheet 1 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	ST	Bx	B
1	0	0		5A1D	0000	00000000	11	0		
2	0	0	CE	5A1D	A5E3	00000000	11	0		
3	0	0		2586	B43A	0000978C	10	0		
4	0	0		2586	8EB4	0000978C	10	0		
5	0	0		1114	D25C	00012F18	9	0		
6	0	0		1114	C148	00012F18	9	0		
7	0	0		1114	B034	00012F18	9	0		
8	0	0		1114	9F20	00012F18	9	0		
9	0	0		1114	8E0C	00012F18	9	0		
10	0	0		080B	F9F0	00025E30	8	0		
11	0	0		080B	F1E5	00025E30	8	0		
12	0	0		080B	E9DA	00025E30	8	0		
13	0	0		080B	E1CF	00025E30	8	0		
14	0	0		080B	D9C4	00025E30	8	0		
15	1	0		080B	D1B9	00025E30	8	0		
16	0	0		17B9	80B0	00327DE0	4	0		
17	0	0		1182	D1EE	0064FBC0	3	0		
18	0	0		1182	C06C	0064FBC0	3	0		
19	0	0		1182	AEEA	0064FBC0	3	0		
20	0	0		1182	9D68	0064FBC0	3	0		
21	0	0		1182	8BE6	0064FBC0	3	0		
22	0	0		0CEF	F4C8	00C9F780	2	0		
23	0	0		0CEF	E7D9	00C9F780	2	0		
24	0	0		0CEF	DAEA	00C9F780	2	0		
25	0	0		0CEF	CDFB	00C9F780	2	0		
26	1	0		0CEF	C10C	00C9F780	2	0		
27	0	0		1518	CEF0	000AB9D0	6	0		65
28	1	0		1518	B9D8	000AB9D0	6	0		
29	0	0		1AA9	A8C0	005AF480	3	0		
30	0	0		1AA9	8E17	005AF480	3	0		
31	0	0		174E	E6DC	00B5E900	2	0		
32	1	0		174E	CF8E	00B5E900	2	0		
33	0	0		1AA9	BA70	00050A00	7	0		5B
34	0	0		1AA9	9FC7	00050A00	7	0		
35	0	0		1AA9	851E	00050A00	7	0		
36	0	0		174E	D4EA	000A1400	6	0		

Table K.7 – Encoder test sequence (sheet 2 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	ST	Bx	B
37	0	0		174E	BD9C	000A1400	6	0		
38	0	0		174E	A64E	000A1400	6	0		
39	0	0		174E	8F00	000A1400	6	0		
40	0	0		1424	EF64	00142800	5	0		
41	0	0		1424	DB40	00142800	5	0		
42	0	0		1424	C71C	00142800	5	0		
43	0	0		1424	B2F8	00142800	5	0		
44	0	0		1424	9ED4	00142800	5	0		
45	0	0		1424	8AB0	00142800	5	0		
46	0	0		119C	ED18	00285000	4	0		
47	0	0		119C	DB7C	00285000	4	0		
48	0	0		119C	C9E0	00285000	4	0		
49	0	0		119C	B844	00285000	4	0		
50	0	0		119C	A6A8	00285000	4	0		
51	0	0		119C	950C	00285000	4	0		
52	0	0		119C	8370	00285000	4	0		
53	0	0		0F6B	E3A8	0050A000	3	0		
54	0	0		0F6B	D43D	0050A000	3	0		
55	0	0		0F6B	C4D2	0050A000	3	0		
56	0	0		0F6B	B567	0050A000	3	0		
57	1	0		0F6B	A5FC	0050A000	3	0		
58	1	0		1424	F6B0	00036910	7	0		51
59	0	0		1AA9	A120	00225CE0	4	0		
60	0	0		1AA9	8677	00225CE0	4	0		
61	0	0		174E	D79C	0044B9C0	3	0		
62	0	0		174E	C04E	0044B9C0	3	0		
63	0	0		174E	A900	0044B9C0	3	0		
64	0	0		174E	91B2	0044B9C0	3	0		
65	0	0		1424	F4C8	00897380	2	0		
66	0	0		1424	E0A4	00897380	2	0		
67	0	0		1424	CC80	00897380	2	0		
68	0	0		1424	B85C	00897380	2	0		
69	0	0		1424	A438	00897380	2	0		
70	0	0		1424	9014	00897380	2	0		
71	1	0		119C	F7E0	0112E700	1	0		
72	1	0		1424	8CE0	001E6A20	6	0		44
73	0	0		1AA9	A120	00F716E0	3	0		

Table K.7 – Encoder test sequence (sheet 3 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	ST	Bx	B
74	1	0		1AA9	8677	00F716E0	3	0		
75	0	0		2516	D548	00041570	8	0		F7
76	1	0		2516	B032	00041570	8	0		
77	0	0		299A	9458	00128230	6	0		
78	0	0		2516	D57C	00250460	5	0		
79	1	0		2516	B066	00250460	5	0		
80	0	0		299A	9458	00963EC0	3	0		
81	1	0		2516	D57C	012C7D80	2	0		
82	0	0		299A	9458	0004B798	8	0		96
83	0	0		2516	D57C	00096F30	7	0		
84	0	0		2516	B066	00096F30	7	0		
85	0	0		2516	8B50	00096F30	7	0		
86	1	0		1EDF	CC74	0012DE60	6	0		
87	1	0		2516	F6F8	009C5FA8	3	0		
88	1	0		299A	9458	0274C628	1	0		
89	0	0		32B4	A668	0004C398	7	0		9D
90	0	0		2E17	E768	00098730	6	0		
91	1	0		2E17	B951	00098730	6	0		
92	0	0		32B4	B85C	002849A8	4	0		
93	1	0		32B4	85A8	002849A8	4	0		
94	0	0		3C3D	CAD0	00A27270	2	0		
95	1	0		3C3D	8E93	00A27270	2	0		
96	0	0		415E	F0F4	00031318	8	0		51
97	1	0		415E	AF96	00031318	8	0		
98	0	0	CE	4639	82BC	000702A0	7	0		
99	1	0		415E	8C72	000E7E46	6	0		
100	0	0	CE	4639	82BC	001D92B4	5	0		
101	1	0		415E	8C72	003B9E6E	4	0		
102	0	0	CE	4639	82BC	0077D304	3	0		
103	1	0		415E	8C72	00F01F0E	2	0		
104	0	0	CE	4639	82BC	01E0D444	1	0		
105	1	0		415E	8C72	0002218E	8	0		78
106	0	0	CE	4639	82BC	0004D944	7	0		
107	1	0		415E	8C72	000A2B8E	6	0		
108	0	0	CE	4639	82BC	0014ED44	5	0		
109	1	0		415E	8C72	002A538E	4	0		
110	0	0	CE	4639	82BC	00553D44	3	0		

Table K.7 – Encoder test sequence (sheet 4 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	ST	Bx	B
111	1	0		415E	8C72	00AAF38E	2	0		
112	0	0	CE	4639	82BC	01567D44	1	0		
113	1	0		415E	8C72	0005738E	8	0		55
114	0	0	CE	4639	82BC	000B7D44	7	0		
115	1	0		415E	8C72	0017738E	6	0		
116	0	0	CE	4639	82BC	002F7D44	5	0		
117	1	0		415E	8C72	005F738E	4	0		
118	0	0	CE	4639	82BC	00BF7D44	3	0		
119	1	0		415E	8C72	017F738E	2	0		
120	0	0	CE	4639	82BC	02FF7D44	1	0		
121	1	0		415E	8C72	0007738E	8	0		BF
122	0	0	CE	4639	82BC	000F7D44	7	0		
123	1	0		415E	8C72	001F738E	6	0		
124	0	0	CE	4639	82BC	003F7D44	5	0		
125	1	0		415E	8C72	007F738E	4	0		
126	0	0	CE	4639	82BC	00FF7D44	3	0		
127	1	0		415E	8C72	01FF738E	2	0		
128	0	0	CE	4639	82BC	03FF7D44	1	0		
129	1	0		415E	8C72	0007738E	8	1		
130	0	0	CE	4639	82BC	000F7D44	7	1		
131	0	0		415E	8C72	001F738E	6	1		
132	0	0		3C3D	9628	003EE71C	5	1		
133	0	0		375E	B3D6	007DCE38	4	1		
134	0	0		32B4	F8F0	00FB9C70	3	1		
135	1	0		32B4	C63C	00FB9C70	3	1		
136	0	0		3C3D	CAD0	03F0BFE0	1	1		
137	1	0		3C3D	8E93	03F0BFE0	1	1		
138	1	0		415E	F0F4	000448D8	7	0		FF00FC
139	0	0	CE	4639	82BC	0009F0DC	6	0		
140	0	0		415E	8C72	00145ABE	5	0		
141	0	0		3C3D	9628	0028B57C	4	0		
142	0	0		375E	B3D6	00516AF8	3	0		
143	0	0		32B4	F8F0	00A2D5F0	2	0		
144	0	0		32B4	C63C	00A2D5F0	2	0		
145	0	0		32B4	9388	00A2D5F0	2	0		
146	0	0		2E17	C1A8	0145ABE0	1	0		

Table K.7 – Encoder test sequence (sheet 5 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	ST	Bx	B
147	1	0		2E17	9391	0145ABE0	1	0		
148	0	0		32B4	B85C	00084568	7	0		51
149	0	0		32B4	85A8	00084568	7	0		
150	0	0		2E17	A5E8	00108AD0	6	0		
151	0	0		299A	EFA2	002115A0	5	0		
152	0	0		299A	C608	002115A0	5	0		
153	0	0		299A	9C6E	002115A0	5	0		
154	0	0		2516	E5A8	00422B40	4	0		
155	0	0		2516	C092	00422B40	4	0		
156	0	0		2516	9B7C	00422B40	4	0		
157	0	0		1EDF	ECCC	00845680	3	0		
158	0	0		1EDF	CDED	00845680	3	0		
159	0	0		1EDF	AF0E	00845680	3	0		
160	0	0		1EDF	902F	00845680	3	0		
161	1	0		1AA9	E2A0	0108AD00	2	0		
162	1	0		2516	D548	000BA7B8	7	0		84
163	1	0		299A	9458	00315FA8	5	0		
164	1	0		32B4	A668	00C72998	3	0		
165	1	0		3C3D	CAD0	031E7530	1	0		
166	1	0		415E	F0F4	000C0F0C	7	0		C7
167	0	0	CE	4639	82BC	00197D44	6	0		
168	0	0		415E	8C72	0033738E	5	0		
169	1	0		3C3D	9628	0066E71C	4	0		
170	1	0		415E	F0F4	019D041C	2	0		
171	0	0	CE	4639	82BC	033B6764	1	0		
172	1	0		415E	8C72	000747CE	8	0		CE
173	0	0	CE	4639	82BC	000F25C4	7	0		
174	1	0		415E	8C72	001EC48E	6	0		
175	1	0	CE	4639	82BC	003E1F44	5	0		
176	1	0		4B85	F20C	00F87D10	3	0		
177	1	0	CE	504F	970A	01F2472E	2	0		
178	0	0	CE	5522	8D76	03E48E5C	1	0		
179	0	0		504F	AA44	00018D60	8	0		F9
180	1	0		4B85	B3EA	00031AC0	7	0		
181	1	0	CE	504F	970A	0007064A	6	0		
182	1	0	CE	5522	8D76	000E0C94	5	0		
183	1	0		59EB	E150	00383250	3	0		

Table K.7 – Encoder test sequence (sheet 6 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	ST	Bx	B
184	0	1		59EB	B3D6	0071736A	2	0		
185	1	0		59EB	B3D6	00E39AAA	1	0		
186	1	1		59EB	B3D6	0007E92A	8	0		38
187	1	1		5522	B3D6	000FD254	7	0		
188	1	1		504F	BD68	001FA4A8	6	0		
189	0	1		4B85	DA32	003F4950	5	0		
190	1	1	CE	504F	970A	007FAFFA	4	0		
191	1	1		4B85	A09E	00FFED6A	3	0		
192	0	1		4639	AA32	01FFDAD4	2	0		
193	0	1	CE	4B85	8C72	04007D9A	1	0		
194	1	1	CE	504F	81DA	0000FB34	8	0	39	00
195	1	1		4B85	A09E	0002597E	7	0		
196	1	1		4639	AA32	0004B2FC	6	0		
197	0	1		415E	C7F2	000965F8	5	0		
198	1	1	CE	4639	82BC	0013D918	4	0		
199	0	1		415E	8C72	00282B36	3	0		
200	0	1	CE	4639	82BC	0050EC94	2	0		
201	1	1		4B85	F20C	0003B250	8	0		28
202	1	1		4B85	A687	0003B250	8	0		
203	1	1		4639	B604	000764A0	7	0		
204	0	1		415E	DF96	000EC940	6	0		
205	1	1	CE	4639	82BC	001ECEF0	5	0		
206	0	1		415E	8C72	003E16E6	4	0		
207	1	1	CE	4639	82BC	007CC3F4	3	0		
208	0	1		415E	8C72	00FA00EE	2	0		
209	1	1	CE	4639	82BC	01F49804	1	0		
210	0	1		415E	8C72	0001A90E	8	0		7D
211	1	1	CE	4639	82BC	0003E844	7	0		
212	0	1		415E	8C72	0008498E	6	0		
213	1	1	CE	4639	82BC	00112944	5	0		
214	0	1		415E	8C72	0022CB8E	4	0		
215	1	1	CE	4639	82BC	00462D44	3	0		
216	1	1		415E	8C72	008CD38E	2	0		
217	1	1		3C3D	9628	0119A71C	1	0		
218	1	1		375E	B3D6	00034E38	8	0		46
219	1	1		32B4	F8F0	00069C70	7	0		
220	1	1		32B4	C63C	00069C70	7	0		

Table K.7 – Encoder test sequence (sheet 7 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	ST	Bx	B
221	0	1		32B4	9388	00069C70	7	0		
222	1	1		3C3D	CAD0	001BF510	5	0		
223	1	1		3C3D	8E93	001BF510	5	0		
224	1	1		375E	A4AC	0037EA20	4	0		
225	0	1		32B4	DA9C	006FD440	3	0		
226	1	1		3C3D	CAD0	01C1F0A0	1	0		
227	1	1		3C3D	8E93	01C1F0A0	1	0		
228	0	1		375E	A4AC	0003E140	8	0		70
229	1	1		3C3D	DD78	00113A38	6	0		
230	0	1		3C3D	A13B	00113A38	6	0		
231	0	1		415E	F0F4	00467CD8	4	0		
232	1	1	CE	4639	82BC	008E58DC	3	0		
233	0	1		415E	8C72	011D2ABE	2	0		
234	1	1	CE	4639	82BC	023AEBA4	1	0		
235	1	1		415E	8C72	0006504E	8	0		8E
236	1	1		3C3D	9628	000CA09C	7	0		
237	1	1		375E	B3D6	00194138	6	0		
238	1	1		32B4	F8F0	00328270	5	0		
239	1	1		32B4	C63C	00328270	5	0		
240	0	1		32B4	9388	00328270	5	0		
241	1	1		3C3D	CAD0	00CB8D10	3	0		
242	1	1		3C3D	8E93	00CB8D10	3	0		
243	1	1		375E	A4AC	01971A20	2	0		
244	0	1		32B4	DA9C	032E3440	1	0		
245	0	1		3C3D	CAD0	000B70A0	7	0		CB
246	1	1		415E	F0F4	002FFCCC	5	0		
247	1	1		415E	AF96	002FFCCC	5	0		
248	1	1		3C3D	DC70	005FF998	4	0		
249	0	1		3C3D	A033	005FF998	4	0		
250	1	1		415E	F0F4	01817638	2	0		
251	0	1		415E	AF96	01817638	2	0		
252	0	1	CE	4639	82BC	0303C8E0	1	0		
253	1	1		4B85	F20C	000F2380	7	0		C0
254	1	1		4B85	A687	000F2380	7	0		
255	0	1		4639	B604	001E4700	6	0		
256	0	1	CE	4B85	8C72	003D6D96	5	0		
Flush:					81DA	007ADB2C	4	0		F6
										FFD9

Table K.8 – Decoder test sequence (sheet 1 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	B
1	0	0		5A1D	0000	655B0000	0	65 5B
2	0	0	CE	5A1D	A5E3	655B0000	0	
3	0	0		2586	B43A	332AA200	7	51
4	0	0		2586	8EB4	332AA200	7	
5	0	0		1114	D25C	66554400	6	
6	0	0		1114	C148	66554400	6	
7	0	0		1114	B034	66554400	6	
8	0	0		1114	9F20	66554400	6	
9	0	0		1114	8E0C	66554400	6	
10	0	0		080B	F9F0	CCAA8800	5	
11	0	0		080B	F1E5	CCAA8800	5	
12	0	0		080B	E9DA	CCAA8800	5	
13	0	0		080B	E1CF	CCAA8800	5	
14	0	0		080B	D9C4	CCAA8800	5	
15	1	0		080B	D1B9	CCAA8800	5	
16	0	0		17B9	80B0	2FC88000	1	
17	0	0		1182	D1EE	5F910000	0	
18	0	0		1182	C06C	5F910000	0	
19	0	0		1182	AEEA	5F910000	0	
20	0	0		1182	9D68	5F910000	0	
21	0	0		1182	8BE6	5F910000	0	
22	0	0		0CEF	F4C8	BF228800	7	44
23	0	0		0CEF	E7D9	BF228800	7	
24	0	0		0CEF	DAEA	BF228800	7	
25	0	0		0CEF	CDFB	BF228800	7	
26	1	0		0CEF	C10C	BF228800	7	
27	0	0		1518	CEF0	B0588000	3	
28	1	0		1518	B9D8	B0588000	3	
29	0	0		1AA9	A8C0	5CC40000	0	
30	0	0		1AA9	8E17	5CC40000	0	
31	0	0		174E	E6DC	B989EE00	7	F7
32	1	0		174E	CF8E	B989EE00	7	
33	0	0		1AA9	BA70	0A4F7000	4	
34	0	0		1AA9	9FC7	0A4F7000	4	
35	0	0		1AA9	851E	0A4F7000	4	
36	0	0		174E	D4EA	149EE000	3	
37	0	0		174E	BD9C	149EE000	3	
38	0	0		174E	A64E	149EE000	3	
39	0	0		174E	8F00	149EE000	3	
40	0	0		1424	EF64	293DC000	2	

Table K.8 – Decoder test sequence (sheet 2 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	B
41	0	0		1424	DB40	293DC000	2	
42	0	0		1424	C71C	293DC000	2	
43	0	0		1424	B2F8	293DC000	2	
44	0	0		1424	9ED4	293DC000	2	
45	0	0		1424	8AB0	293DC000	2	
46	0	0		119C	ED18	527B8000	1	
47	0	0		119C	DB7C	527B8000	1	
48	0	0		119C	C9E0	527B8000	1	
49	0	0		119C	B844	527B8000	1	
50	0	0		119C	A6A8	527B8000	1	
51	0	0		119C	950C	527B8000	1	
52	0	0		119C	8370	527B8000	1	
53	0	0		0F6B	E3A8	A4F70000	0	
54	0	0		0F6B	D43D	A4F70000	0	
55	0	0		0F6B	C4D2	A4F70000	0	
56	0	0		0F6B	B567	A4F70000	0	
57	1	0		0F6B	A5FC	A4F70000	0	
58	1	0		1424	F6B0	E6696000	4	96
59	0	0		1AA9	A120	1EEB0000	1	
60	0	0		1AA9	8677	1EEB0000	1	
61	0	0		174E	D79C	3DD60000	0	
62	0	0		174E	C04E	3DD60000	0	
63	0	0		174E	A900	3DD60000	0	
64	0	0		174E	91B2	3DD60000	0	
65	0	0		1424	F4C8	7BAD3A00	7	9D
66	0	0		1424	E0A4	7BAD3A00	7	
67	0	0		1424	CC80	7BAD3A00	7	
68	0	0		1424	B85C	7BAD3A00	7	
69	0	0		1424	A438	7BAD3A00	7	
70	0	0		1424	9014	7BAD3A00	7	
71	1	0		119C	F7E0	F75A7400	6	
72	1	0		1424	8CE0	88B3A000	3	
73	0	0		1AA9	A120	7FBD0000	0	
74	1	0		1AA9	8677	7FBD0000	0	
75	0	0		2516	D548	9F7A8800	5	51
76	1	0		2516	B032	9F7A8800	5	
77	0	0		299A	9458	517A2000	3	
78	0	0		2516	D57C	A2F44000	2	
79	1	0		2516	B066	A2F44000	2	
80	0	0		299A	9458	5E910000	0	

Table K.8 – Decoder test sequence (sheet 3 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	B
81	1	0		2516	D57C	BD22F000	7	78
82	0	0		299A	9458	32F3C000	5	
83	0	0		2516	D57C	65E78000	4	
84	0	0		2516	B066	65E78000	4	
85	0	0		2516	8B50	65E78000	4	
86	1	0		1EDF	CC74	CBCF0000	3	
87	1	0		2516	F6F8	F1D00000	0	
88	1	0		299A	9458	7FB95400	6	55
89	0	0		32B4	A668	53ED5000	4	
90	0	0		2E17	E768	A7DAA000	3	
91	1	0		2E17	B951	A7DAA000	3	
92	0	0		32B4	B85C	72828000	1	
93	1	0		32B4	85A8	72828000	1	
94	0	0		3C3D	CAD0	7E3B7E00	7	BF
95	1	0		3C3D	8E93	7E3B7E00	7	
96	0	0		415E	F0F4	AF95F800	5	
97	1	0		415E	AF96	AF95F800	5	
98	0	0	CE	4639	82BC	82BBF000	4	
99	1	0		415E	8C72	8C71E000	3	
100	0	0	CE	4639	82BC	82BBC000	2	
101	1	0		415E	8C72	8C718000	1	
102	0	0	CE	4639	82BC	82BB0000	0	
103	1	0		415E	8C72	8C71FE00	7	FF 00
104	0	0	CE	4639	82BC	82BBFC00	6	
105	1	0		415E	8C72	8C71F800	5	
106	0	0	CE	4639	82BC	82BBF000	4	
107	1	0		415E	8C72	8C71E000	3	
108	0	0	CE	4639	82BC	82BBC000	2	
109	1	0		415E	8C72	8C718000	1	
110	0	0	CE	4639	82BC	82BB0000	0	
111	1	0		415E	8C72	8C71F800	7	FC
112	0	0	CE	4639	82BC	82BBF000	6	
113	1	0		415E	8C72	8C71E000	5	
114	0	0	CE	4639	82BC	82BBC000	4	
115	1	0		415E	8C72	8C718000	3	
116	0	0	CE	4639	82BC	82BB0000	2	
117	1	0		415E	8C72	8C700000	1	
118	0	0	CE	4639	82BC	82B80000	0	
119	1	0		415E	8C72	8C6AA200	7	51
120	0	0	CE	4639	82BC	82AD4400	6	

Table K.8 – Decoder test sequence (sheet 4 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	B
121	1	0		415E	8C72	8C548800	5	
122	0	0	CE	4639	82BC	82811000	4	
123	1	0		415E	8C72	8BFC2000	3	
124	0	0	CE	4639	82BC	81D04000	2	
125	1	0		415E	8C72	8A9A8000	1	
126	0	0	CE	4639	82BC	7F0D0000	0	
127	1	0		415E	8C72	85150800	7	84
128	0	0	CE	4639	82BC	74021000	6	
129	1	0		415E	8C72	6EFE2000	5	
130	0	0	CE	4639	82BC	47D44000	4	
131	0	0		415E	8C72	16A28000	3	
132	0	0		3C3D	9628	2D450000	2	
133	0	0		375E	B3D6	5A8A0000	1	
134	0	0		32B4	F8F0	B5140000	0	
135	1	0		32B4	C63C	B5140000	0	
136	0	0		3C3D	CAD0	86331C00	6	C7
137	1	0		3C3D	8E93	86331C00	6	
138	1	0		415E	F0F4	CF747000	4	
139	0	0	CE	4639	82BC	3FBCE000	3	
140	0	0		415E	8C72	0673C000	2	
141	0	0		3C3D	9628	0CE78000	1	
142	0	0		375E	B3D6	19CF0000	0	
143	0	0		32B4	F8F0	339F9C00	7	CE
144	0	0		32B4	C63C	339F9C00	7	
145	0	0		32B4	9388	339F9C00	7	
146	0	0		2E17	C1A8	673F3800	6	
147	1	0		2E17	9391	673F3800	6	
148	0	0		32B4	B85C	0714E000	4	
149	0	0		32B4	85A8	0714E000	4	
150	0	0		2E17	A5E8	0E29C000	3	
151	0	0		299A	EFA2	1C538000	2	
152	0	0		299A	C608	1C538000	2	
153	0	0		299A	9C6E	1C538000	2	
154	0	0		2516	E5A8	38A70000	1	
155	0	0		2516	C092	38A70000	1	
156	0	0		2516	9B7C	38A70000	1	
157	0	0		1EDF	ECCC	714E0000	0	
158	0	0		1EDF	CDED	714E0000	0	
159	0	0		1EDF	AF0E	714E0000	0	
160	0	0		1EDF	902F	714E0000	0	

Table K.8 – Decoder test sequence (sheet 5 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	B
161	1	0		1AA9	E2A0	E29DF200	7	F9
162	1	0		2516	D548	D5379000	4	
163	1	0		299A	9458	94164000	2	
164	1	0		32B4	A668	A5610000	0	
165	1	0		3C3D	CAD0	C6B4E400	6	39
166	1	0		415E	F0F4	E0879000	4	
167	0	0	CE	4639	82BC	61E32000	3	
168	0	0		415E	8C72	4AC04000	2	
169	1	0		3C3D	9628	95808000	1	
170	1	0		415E	F0F4	EE560000	7	00
171	0	0	CE	4639	82BC	7D800000	6	
172	1	0		415E	8C72	81FA0000	5	
173	0	0	CE	4639	82BC	6DCC0000	4	
174	1	0		415E	8C72	62920000	3	
175	1	0	CE	4639	82BC	2EFC0000	2	
176	1	0		4B85	F20C	BBF00000	0	
177	1	0	CE	504F	970A	2AD25000	7	28
178	0	0	CE	5522	8D76	55A4A000	6	
179	0	0		504F	AA44	3AA14000	5	
180	1	0		4B85	B3EA	75428000	4	
181	1	0	CE	504F	970A	19BB0000	3	
182	1	0	CE	5522	8D76	33760000	2	
183	1	0		59EB	E150	CDD80000	0	
184	0	1		59EB	B3D6	8CE6FA00	7	7D
185	1	0		59EB	B3D6	65F7F400	6	
186	1	1		59EB	B3D6	1819E800	5	
187	1	1		5522	B3D6	3033D000	4	
188	1	1		504F	BD68	6067A000	3	
189	0	1		4B85	DA32	C0CF4000	2	
190	1	1	CE	504F	970A	64448000	1	
191	1	1		4B85	A09E	3B130000	0	
192	0	1		4639	AA32	76268C00	7	46
193	0	1	CE	4B85	8C72	245B1800	6	
194	1	1	CE	504F	81DA	48B63000	5	
195	1	1		4B85	A09E	2E566000	4	
196	1	1		4639	AA32	5CACC000	3	
197	0	1		415E	C7F2	B9598000	2	
198	1	1	CE	4639	82BC	658B0000	1	
199	0	1		415E	8C72	52100000	0	
200	0	1	CE	4639	82BC	0DF8E000	7	70

Table K.8 – Decoder test sequence (sheet 6 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	B
201	1	1		4B85	F20C	37E38000	5	
202	1	1		4B85	A687	37E38000	5	
203	1	1		4639	B604	6FC70000	4	
204	0	1		415E	DF96	DF8E0000	3	
205	1	1	CE	4639	82BC	82AC0000	2	
206	0	1		415E	8C72	8C520000	1	
207	1	1	CE	4639	82BC	827C0000	0	
208	0	1		415E	8C72	8BF31C00	7	8E
209	1	1	CE	4639	82BC	81BE3800	6	
210	0	1		415E	8C72	8A767000	5	
211	1	1	CE	4639	82BC	7EC4E000	4	
212	0	1		415E	8C72	8483C000	3	
213	1	1	CE	4639	82BC	72DF8000	2	
214	0	1		415E	8C72	6CB90000	1	
215	1	1	CE	4639	82BC	434A0000	0	
216	1	1		415E	8C72	0D8F9600	7	CB
217	1	1		3C3D	9628	1B1F2C00	6	
218	1	1		375E	B3D6	363E5800	5	
219	1	1		32B4	F8F0	6C7CB000	4	
220	1	1		32B4	C63C	6C7CB000	4	
221	0	1		32B4	9388	6C7CB000	4	
222	1	1		3C3D	CAD0	2EA2C000	2	
223	1	1		3C3D	8E93	2EA2C000	2	
224	1	1		375E	A4AC	5D458000	1	
225	0	1		32B4	DA9C	BA8B0000	0	
226	1	1		3C3D	CAD0	4A8F0000	6	C0
227	1	1		3C3D	8E93	4A8F0000	6	
228	0	1		375E	A4AC	951E0000	5	
229	1	1		3C3D	DD78	9F400000	3	
230	0	1		3C3D	A13B	9F400000	3	
231	0	1		415E	F0F4	E9080000	1	
232	1	1	CE	4639	82BC	72E40000	0	
233	0	1		415E	8C72	6CC3EC00	7	F6
234	1	1	CE	4639	82BC	435FD800	6	
235	1	1		415E	8C72	0DB9B000	5	
236	1	1		3C3D	9628	1B736000	4	
237	1	1		375E	B3D6	36E6C000	3	
238	1	1		32B4	F8F0	6DCD8000	2	
239	1	1		32B4	C63C	6DCD8000	2	
240	0	1		32B4	9388	6DCD8000	2	

Table K.8 – Decoder test sequence (sheet 7 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	B
241	1	1		3C3D	CAD0	33E60000	0	
242	1	1		3C3D	8E93	33E60000	0	
Marker detected: zero byte fed to decoder								
243	1	1		375E	A4AC	67CC0000	7	
244	0	1		32B4	DA9C	CF980000	6	
245	0	1		3C3D	CAD0	9EC00000	4	
246	1	1		415E	F0F4	40B40000	2	
247	1	1		415E	AF96	40B40000	2	
248	1	1		3C3D	DC70	81680000	1	
249	0	1		3C3D	A033	81680000	1	
Marker detected: zero byte fed to decoder								
250	1	1		415E	F0F4	75C80000	7	
251	0	1		415E	AF96	75C80000	7	
252	0	1	CE	4639	82BC	0F200000	6	
253	1	1		4B85	F20C	3C800000	4	
254	1	1		4B85	A687	3C800000	4	
255	0	1		4639	B604	79000000	3	
256	0	1	CE	4B85	8C72	126A0000	2	

K.5 Low-pass downsampling filters for hierarchical coding

In this section simple examples are given of downsampling filters which are compatible with the upsampling filter defined in J.1.1.2.

Figure K.5 shows the weighting of neighbouring samples for simple one-dimensional horizontal and vertical low-pass filters. The output of the filter must be normalized by the sum of the neighbourhood weights.

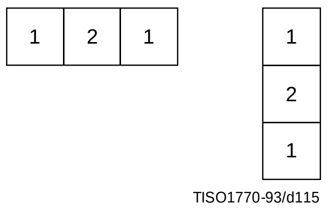


Figure K.5 – Low-pass filter example

The centre sample in Figure K.5 should be aligned with the left column or top line of the high resolution image when calculating the left column or top line of the low resolution image. Sample values which are situated outside of the image boundary are replicated from the sample values at the boundary to provide missing edge values.

If the image being downsampled has an odd width or length, the odd dimension is increased by 1 by sample replication on the right edge or bottom line before downsampling.

K.6 Domain of applicability of DCT and spatial coding techniques

The DCT coder is intended for lossy coding in a range from quite visible loss to distortion well below the threshold for visibility. However in general, DCT-based processes cannot be used for true lossless coding.

The lossless coder is intended for completely lossless coding. The lossless coding process is significantly less effective than the DCT-based processes for distortions near and above the threshold of visibility.

The point transform of the input to the lossless coder permits a very restricted form of lossy coding with the "lossless" coder. (The coder is still lossless after the input point transform.) Since the DCT is intended for lossy coding, there may be some confusion about when this alternative lossy technique should be used.

Lossless coding with a point transformed input is intended for applications which cannot be addressed by DCT coding techniques. Among these are

– true lossless coding to a specified precision;

– lossy coding with precisely defined error bounds;

– hierarchical progression to a truly lossless final stage.

If lossless coding with a point transformed input is used in applications which can be met effectively by DCT coding, the results will be significantly less satisfactory. For example, distortion in the form of visible contours usually appears when precision of the luminance component is reduced to about six bits. For normal image data, this occurs at bit rates well above those for which the DCT gives outputs which are visually indistinguishable from the source.

K.7 Domain of applicability of the progressive coding modes of operation

Two very different progressive coding modes of operation have been defined, progressive coding of the DCT coefficients and hierarchical progression. Progressive coding of the DCT coefficients has two complementary procedures, spectral selection and successive approximation. Because of this diversity of choices, there may be some confusion as to which method of progression to use for a given application.

K.7.1 Progressive coding of the DCT

In progressive coding of the DCT coefficients two complementary procedures are defined for decomposing the 8×8 DCT coefficient array, spectral selection and successive approximation. Spectral selection partitions zig-zag array of DCT coefficients into "bands", one band being coded in each scan. Successive approximation codes the coefficients with reduced precision in the first scan; in each subsequent scan the precision is increased by one bit.

A single forward DCT is calculated for these procedures. When all coefficients are coded to full precision, the DCT is the same as in the sequential mode. Therefore, like the sequential DCT coding, progressive coding of DCT coefficients is intended for applications which need very good compression for a given level of visual distortion.

The simplest progressive coding technique is spectral selection; indeed, because of this simplicity, some applications may choose – despite the limited progression that can be achieved – to use only spectral selection. Note, however, that the absence of high frequency bands typically leads – for a given bit rate – to a significantly lower image quality in the intermediate stages than can be achieved with the more general progressions. The net coding efficiency at the completion of the final stage is typically comparable to or slightly less than that achieved with the sequential DCT.

A much more flexible progressive system is attained at some increase in complexity when successive approximation is added to the spectral selection progression. For a given bit rate, this system typically provides significantly better image quality than spectral selection alone. The net coding efficiency at the completion of the final stage is typically comparable to or slightly better than that achieved with the sequential DCT.

K.7.2 Hierarchical progression

Hierarchical progression permits a sequence of outputs of increasing spatial resolution, and also allows refinement of image quality at a given spatial resolution. Both DCT and spatial versions of the hierarchical progression are allowed, and progressive coding of DCT coefficients may be used in a frame of the DCT hierarchical progression.

The DCT hierarchical progression is intended for applications which need very good compression for a given level of visual distortion; the spatial hierarchical progression is intended for applications which need a simple progression with a truly lossless final stage. Figure K.6 illustrates examples of these two basic hierarchical progressions.

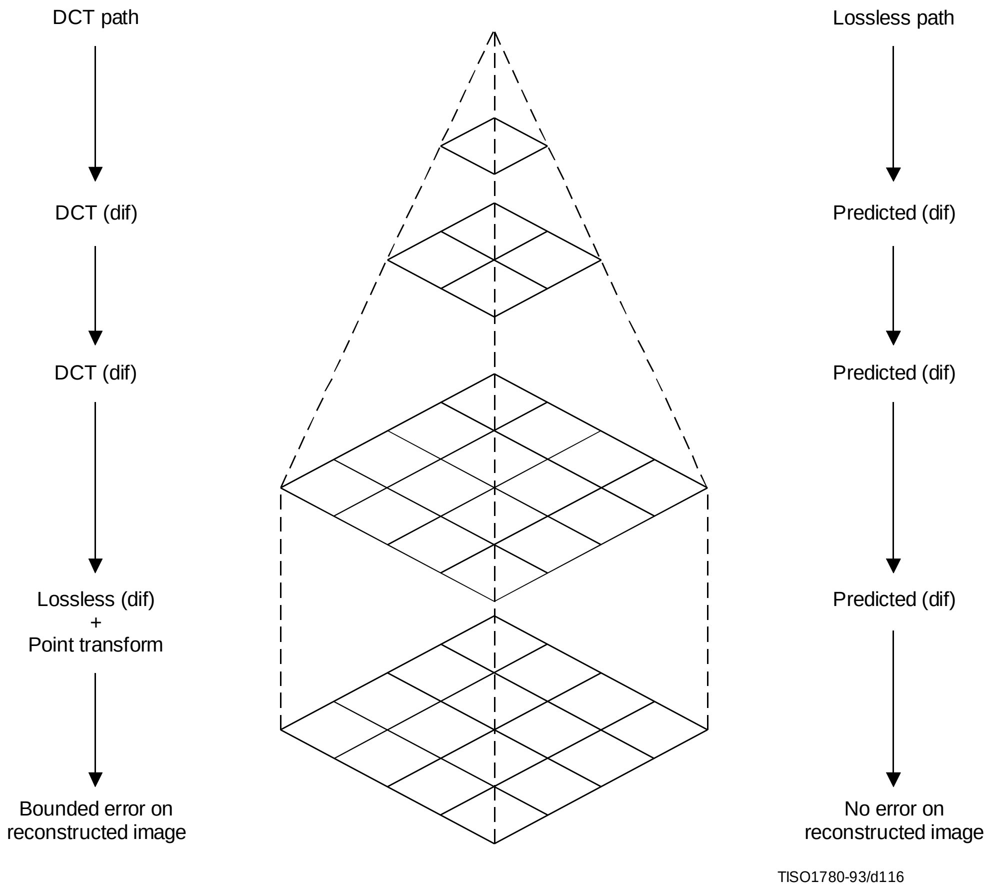


Figure K.6 – Sketch of the basic operations of the hierarchical mode

K.7.2.1 DCT Hierarchical progression

If a DCT hierarchical progression uses reduced spatial resolution, the early stages of the progression can have better image quality for a given bit rate than the early stages of non-hierarchical progressive coding of the DCT coefficients. However, at the point where the distortion between source and output becomes indistinguishable, the coding efficiency achieved with a DCT hierarchical progression is typically significantly lower than the coding efficiency achieved with a non-hierarchical progressive coding of the DCT coefficients.

While the hierarchical DCT progression is intended for lossy progressive coding, a final spatial differential coding stage can be used. When this final stage is used, the output can be almost lossless, limited only by the difference between the encoder and decoder IDCT implementations. Since IDCT implementations can differ significantly, truly lossless coding after a DCT hierarchical progression cannot be guaranteed. An important alternative, therefore, is to use the input point transform of the final lossless differential coding stage to reduce the precision of the differential input. This allows a bounding of the difference between source and output at a significantly lower cost in coded bits than coding of the full precision spatial difference would require.

K.7.2.2 Spatial hierarchical progression

If lossless progression is required, a very simple hierarchical progression may be used in which the spatial lossless coder with point transformed input is used as a first stage. This first stage is followed by one or more spatial differential coding stages. The first stage should be nearly lossless, such that the low order bits which are truncated by the point transform are essentially random – otherwise the compression efficiency will be degraded relative to non-progressive lossless coding.

K.8 Suppression of block-to-block discontinuities in decoded images

A simple technique is available for suppressing the block-to-block discontinuities which can occur in images compressed by DCT techniques.

The first few (five in this example) low frequency DCT coefficients are predicted from the nine DC values of the block and the eight nearest-neighbour blocks, and the predicted values are used to suppress blocking artifacts in smooth areas of the image.

The prediction equations for the first five AC coefficients in the zig-zag sequence are obtained as follows:

K.8.1 AC prediction

The sample field in a 3 by 3 array of blocks (each block containing an 8×8 array of samples) is modeled by a two-dimensional second degree polynomial of the form:

$$P(x,y) = A1(x^2y^2) + A2(x^2y) + A3(xy^2) + A4(x^2) + A5(xy) + A6(y^2) + A7(x) + A8(y) + A9$$

The nine coefficients A1 through A9 are uniquely determined by imposing the constraint that the mean of P(x,y) over each of the nine blocks must yield the correct DC-values.

Applying the DCT to the quadratic field predicting the samples in the central block gives a prediction of the low frequency AC coefficients depicted in Figure K.7.

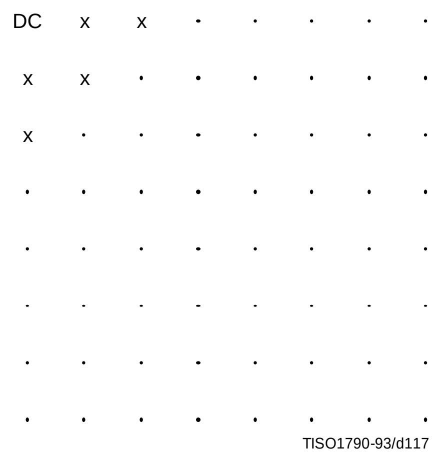


Figure K.7 – DCT array positions of predicted AC coefficients

The prediction equations derived in this manner are as follows:

For the two dimensional array of DC values shown

$$\begin{array}{lll} DC_1 & DC_2 & DC_3 \\ DC_4 & DC_5 & DC_6 \\ DC_7 & DC_8 & DC_9 \end{array}$$

The unquantized prediction equations are

$$AC_{01} = 1{,}13885\ (DC_4 - DC_6)$$
$$AC_{10} = 1{,}13885\ (DC_2 - DC_8)$$
$$AC_{20} = 0{,}27881\ (DC_2 + DC_8 - 2 \times DC_5)$$
$$AC_{11} = 0{,}16213\ ((DC_1 - DC_3) - (DC_7 - DC_9))$$
$$AC_{02} = 0{,}27881\ (DC_4 + DC_6 - 2 \times DC_5)$$

The scaling of the predicted AC coefficients is consistent with the DCT normalization defined in A.3.3.

K.8.2 Quantized AC prediction

The prediction equations can be mapped to a form which uses quantized values of the DC coefficients and which computes quantized AC coefficients using integer arithmetic. The quantized DC coefficients need to be scaled, however, such that the predicted coefficients have fractional bit precision.

First, the prediction equation coefficients are scaled by 32 and rounded to the nearest integer. Thus,

$$1,13885 \times 32 = 36$$

$$0,27881 \times 32 = 9$$

$$0,16213 \times 32 = 5$$

The multiplicative factors are then scaled by the ratio of the DC and AC quantization factors and rounded appropriately. The normalization defined for the DCT introduces another factor of 8 in the unquantized DC values. Therefore, in terms of the quantized DC values, the predicted quantized AC coefficients are given by the equations below. Note that if (for example) the DC values are scaled by a factor of 4, the AC predictions will have 2 fractional bits of precision relative to the quantized DCT coefficients.

$$QAC_{01} = ((R_d \times Q_{01}) + (36 \times Q_{00} \times (QDC_4 - QDC_6)))/(256 \times Q_{01})$$
$$QAC_{10} = ((R_d \times Q_{10}) + (36 \times Q_{00} \times (QDC_2 - QDC_8)))/(256 \times Q_{10})$$
$$QAC_{20} = ((R_d \times Q_{20}) + (9 \times Q_{00} \times (QDC_2 + QDC_8 - 2 \times QDC_5)))/(256 \times Q_{20})$$
$$QAC_{11} = ((R_d \times Q_{11}) + (5 \times Q_{00} \times ((QDC_1 - QDC_3) - (QDC_7 - QDC_9))))/(256 \times Q_{11})$$
$$QAC_{02} = ((R_d \times Q_{02}) + (9 \times Q_{00} \times (QDC_4 + QDC_6 - 2 \times QDC_5)))/(256 \times Q_{02})$$

where QDC_x and QAC_{xy} are the quantized and scaled DC and AC coefficient values. The constant Rd is added to get a correct rounding in the division. Rd is 128 for positive numerators, and –128 for negative numerators.

Predicted values should not override coded values. Therefore, predicted values for coefficients which are already non-zero should be set to zero. Predictions should be clamped if they exceed a value which would be quantized to a non-zero value for the current precision in the successive approximation.

K.9 Modification of dequantization to improve displayed image quality

For a progression where the first stage successive approximation bit, Al, is set to 3, uniform quantization of the DCT gives the following quantization and dequantization levels for a sequence of successive approximation scans, as shown in Figure K.8:

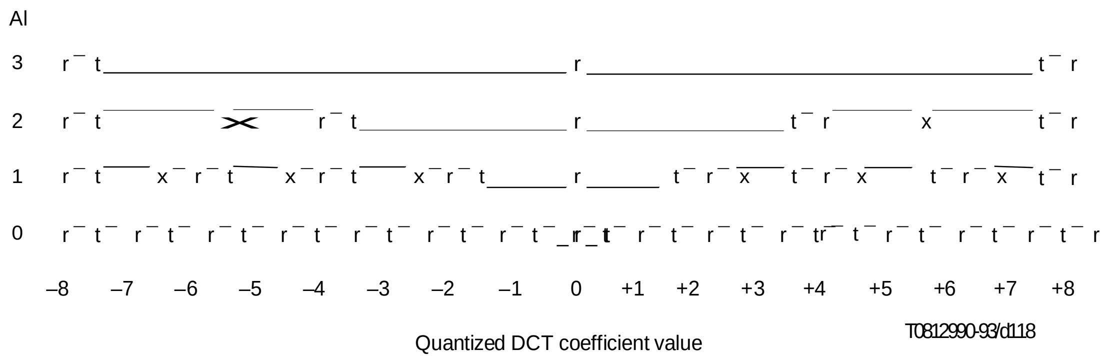


Figure K.8 – Illustration of two reconstruction strategies

The column to the left labelled "Al" gives the bit position specified in the scan header. The quantized DCT coefficient magnitudes are therefore divided by 2^{Al} during that scan.

Referring to the final scan (Al = 0), the points marked with “t” are the threshold values, while the points marked with “r” are the reconstruction values. The unquantized output is obtained by multiplying the horizontal scale in Figure K.8 by the quantization value.

The quantization interval for a coefficient value of zero is indicated by the depressed interval of the line. As the bit position Al is increased, a “fat zero” quantization interval develops around the zero DCT coefficient value. In the limit where the scaling factor is very large, the zero interval is twice as large as the rest of the quantization intervals.

Two different reconstruction strategies are shown. The points marked “r” are the reconstruction obtained using the normal rounding rules for the DCT for the complete full precision output. This rule seems to give better image quality when high bandwidth displays are used. The points marked “x” are an alternative reconstruction which tends to give better images on lower bandwidth displays. “x” and “r” are the same for slice 0. The system designer must determine which strategy is best for the display system being used.

K.10 Example of point transform

The difference between the arithmetic-shift-right by Pt and divide by 2^{Pt} can be seen from the following:

After the level shift the DC has values from +127 to –128. Consider values near zero (after the level shift), and the case where Pt = 1:

Before level shift	Before point transform	After divide by 2	After shift-right-arithmetic 1
131	+3	+1	+1
130	+2	+1	+1
129	+1	0	0
128	0	0	0
127	–1	0	–1
126	–2	–1	–1
125	–3	–1	–2
124	–4	–2	–2
123	–5	–2	–3

The key difference is in the truncation of precision. The divide truncates the magnitude; the arithmetic shift truncates the LSB. With a divide by 2 we would get non-uniform quantization of the DC values; therefore we use the shift-right-arithmetic operation.

For positive values, the divide by 2 and the shift-right-arithmetic by 1 operations are the same. Therefore, the shift-right-arithmetic by 1 operation effectively is a divide by 2 when the point transform is done before the level shift.

Annex L

Patents

(This annex does not form an integral part of this Recommendation | International Standard)

L.1 Introductory remarks

The user's attention is called to the possibility that – for some of the coding processes specified in Annexes F, G, H, and J – compliance with this Specification may require use of an invention covered by patent rights.

By publication of this Specification, no position is taken with respect to the validity of this claim or of any patent rights in connection therewith. However, for each patent listed in this annex, the patent holder has filed with the Information Technology Task Force (ITTF) and the Telecommunication Standardization Bureau (TSB) a statement of willingness to grant a license under these rights on reasonable and non-discriminatory terms and conditions to applicants desiring to obtain such a license.

The criteria for including patents in this annex are:

a) the patent has been identified by someone who is familiar with the technical fields relevant to this Specification, and who believes use of the invention covered by the patent is *required* for implementation of one or more of the coding processes specified in Annexes F, G, H, or J;

b) the patent-holder has written a letter to the ITTF and TSB, stating willingness to grant a license to an unlimited number of applicants throughout the world under reasonable terms and conditions that are demonstrably free of any unfair discrimination.

This list of patents shall be updated, if necessary, upon publication of any revisions to the Recommendation | International Standard.

L.2 List of patents

The following patents may be required for implementation of any one of the processes specified in Annexes F, G, H, and J which uses arithmetic coding:

US 4,633,490, December 30, 1986, IBM, MITCHELL (J.L.) and GOERTZEL (G.): *Symmetrical Adaptive Data Compression/Decompression System.*

US 4,652,856, February 4, 1986, IBM, MOHIUDDIN (K.M.) and RISSANEN (J.J.): *A Multiplication-free Multi-Alphabet Arithmetic Code.*

US 4,369,463, January 18, 1983, IBM, ANASTASSIOU (D.) and MITCHELL (J.L.): *Grey Scale Image Compression with Code Words a Function of Image History*.

US 4,749,983, June 7, 1988, IBM, LANGDON (G.): *Compression of Multilevel Signals*.

US 4,935,882, June 19, 1990, IBM, PENNEBAKER (W.B.) and MITCHELL (J.L.): *Probability Adaptation for Arithmetic Coders*.

US 4,905,297, February 27, 1990, IBM, LANGDON (G.G.), Jr., MITCHELL (J.L.), PENNEBAKER (W.B.), and RISSANEN (J.J.): *Arithmetic Coding Encoder and Decoder System*.

US 4,973,961, November 27, 1990, AT&T, CHAMZAS (C.), DUTTWEILER (D.L.): *Method and Apparatus for Carry-over Control in Arithmetic Entropy Coding*.

US 5,025,258, June 18, 1991, AT&T, DUTTWEILER (D.L): *Adaptive Probability Estimator for Entropy Encoding/Decoding*.

US 5,099,440, March 24, 1992, IBM, PENNEBAKER (W.B.) and MITCHELL (J.L.): *Probability Adaptation for Arithmetic Coders*.

Japanese Patent Application 2-46275, February 26, 1990, MEL ONO (F.), KIMURA (T.), YOSHIDA (M.), and KINO (S.): *Coding System*.

The following patent may be required for implementation of any one of the hierarchical processes specified in Annex H when used with a lossless final frame:

US 4,665,436, May 12, 1987, EI OSBORNE (J.A.) and SEIFFERT (C.): *Narrow Bandwidth Signal Transmission*.

No other patents required for implementation of any of the other processes specified in Annexes F, G, H, or J had been identified at the time of publication of this Specification.

L.3 Contact addresses for patent information

Director, Telecommunication Standardization Bureau (formerly CCITT)
International Telecommunication Union
Place des Nations
CH-1211 Genève 20, Switzerland
Tel. +41 (22) 730 5111
Fax: +41 (22) 730 5853

Information Technology Task Force
International Organization for Standardization
1, rue de Varembé
CH-1211 Genève 20, Switzerland
Tel: +41 (22) 734 0150
Fax: +41 (22) 733 3843

Program Manager, Licensing
Intellectual Property and Licensing Services
IBM Corporation
208 Harbor Drive
P.O. Box 10501
Stamford, Connecticut 08904-2501, USA
Tel: +1 (203) 973 7935
Fax: +1 (203) 973 7981 or +1 (203) 973 7982

Mitsubishi Electric Corp.
Intellectual Property License Department
1-2-3 Morunouchi, Chiyoda-ku
Tokyo 100, Japan
Tel: +81 (3) 3218 3465
Fax: +81 (3) 3215 3842

AT&T Intellectual Property Division Manager
Room 3A21
10 Independence Blvd.
Warren, NJ 07059, USA
Tel: +1 (908) 580 5392
Fax: +1 (908) 580 6355

Senior General Manager
Corporate Intellectual Property and Legal Headquarters
Canon Inc.
30-2 Shimomaruko 3-chome
Ohta-ku Tokyo 146 Japan
Tel: +81 (3) 3758 2111
Fax: +81 (3) 3756 0947

Chief Executive Officer
Electronic Imagery, Inc.
1100 Park Central Boulevard South
Suite 3400
Pompano Beach, FL 33064, USA
Tel: +1 (305) 968 7100
Fax: +1 (305) 968 7319

Annex M

Bibliography

(This annex does not form an integral part of this Recommendation | International Standard)

M.1 General references

LEGER (A.), OMACHI (T.), and WALLACE (G.K.): JPEG Still Picture Compression Algorithm, *Optical Engineering*, Vol. 30, No. 7, pp. 947-954, 1991.

RABBANI (M.) and JONES (P.): Digital Image Compression Techniques, *Tutorial Texts in Optical Engineering*, Vol. TT7, SPIE Press, 1991.

HUDSON (G.), YASUDA (H.) and SEBESTYEN (I.): The International Standardization of a Still Picture Compression Technique, *Proc. of IEEE Global Telecommunications Conference*, pp. 1016-1021, 1988.

LEGER (A.), MITCHELL (J.) and YAMAZAKI (Y.): Still Picture Compression Algorithm Evaluated for International Standardization, *Proc. of the IEEE Global Telecommunications Conference*, pp. 1028-1032, 1988.

WALLACE (G.), VIVIAN (R.) and POULSEN (H.): Subjective Testing Results for Still Picture Compression Algorithms for International Standardization, *Proc. of the IEEE Global Telecommunications Conference*, pp. 1022-1027, 1988.

MITCHELL (J.L.) and PENNEBAKER (W.B.): Evolving JPEG Colour Data Compression Standard, *Standards for Electronic Imaging Systems*, M. Nier, M.E. Courtot, Editors, SPIE, Vol. CR37, pp. 68-97, 1991.

WALLACE (G.K.): The JPEG Still Picture Compression Standard, *Communications of the ACM*, Vol. 34, No. 4, pp. 31-44, 1991.

NETRAVALI (A.N.) and HASKELL (B.G.): *Digital Pictures: Representation and Compression*, Plenum Press, New York 1988.

PENNEBAKER (W.B.) and MITCHELL (J.L.): *JPEG: Still Image Data Compression Standard*, Van Nostrand Reinhold, New York 1993.

M.2 DCT references

CHEN (W.), SMITH (C.H.) and FRALICK (S.C.): A Fast Computational Algorithm for the Discrete Cosine Transform, *IEEE Trans. on Communications*, Vol. COM-25, pp. 1004-1009, 1977.

AHMED (N.), NATARAJAN (T.) and RAO (K.R.): Discrete Cosine Transform, *IEEE Trans. on Computers*, Vol. C-23, pp. 90-93, 1974.

NARASINHA (N.J.) and PETERSON (A.M.): On the Computation of the Discrete Cosine Transform, *IEEE Trans. on Communications*, Vol. COM-26, No. 6, pp. 966-968, 1978.

DUHAMEL (P.) and GUILLEMOT (C.): Polynomial Transform Computation of the 2-D DCT, *Proc. IEEE ICASSP-90*, pp. 1515-1518, Albuquerque, New Mexico 1990.

FEIG (E.): A Fast Scaled DCT Algorithm, in *Image Processing Algorithms and Techniques*, Proc. SPIE, Vol. 1244, K.S. Pennington and R. J. Moorhead II, Editors, pp. 2-13, Santa Clara, California, 1990.

HOU (H.S.): A Fast Recursive Algorithm for Computing the Discrete Cosine Transform, *IEEE Trans. Acoust. Speech and Signal Processing*, Vol. ASSP-35, No. 10, pp. 1455-1461.

LEE (B.G.): A New Algorithm to Compute the Discrete Cosine Transform, *IEEE Trans. on Acoust., Speech and Signal Processing*, Vol. ASSP-32, No. 6, pp. 1243-1245, 1984.

LINZER (E.N.) and FEIG (E.): New DCT and Scaled DCT Algorithms for Fused Multiply/Add Architectures, *Proc. IEEE ICASSP-91*, pp. 2201-2204, Toronto, Canada, 1991.

VETTERLI (M.) and NUSSBAUMER (H.J.): Simple FFT and DCT Algorithms with Reduced Number of Operations, *Signal Processing*, 1984.

VETTERLI (M.): Fast 2-D Discrete Cosine Transform, *Proc. IEEE ICASSP-85,* pp. 1538-1541, Tampa, Florida, 1985.

ARAI (Y.), AGUI (T.), and NAKAJIMA (M.): A Fast DCT-SQ Scheme for Images, *Trans. of IEICE,* Vol. E.71, No. 11, pp. 1095-1097, 1988.

SUEHIRO (N.) and HATORI (M.): Fast Algorithms for the DFT and other Sinusoidal Transforms, *IEEE Trans. on Acoust., Speech and Signal Processing,* Vol ASSP-34, No. 3, pp. 642-644, 1986.

M.3 Quantization and human visual model references

CHEN (W.H.) and PRATT (W.K.): Scene adaptive coder, *IEEE Trans. on Communications*, Vol. COM-32, pp. 225-232, 1984.

GRANRATH (D.J.): The role of human visual models in image processing, *Proceedings of the IEEE,* Vol. 67, pp. 552-561, 1981.

LOHSCHELLER (H.): Vision adapted progressive image transmission, *Proceedings of EUSIPCO,* Vol. 83, pp. 191-194, 1983.

LOHSCHELLER (H.) and FRANKE (U.): Colour picture coding – Algorithm optimization and technical realization, *Frequenze,* Vol. 41, pp. 291-299, 1987.

LOHSCHELLER (H.): A subjectively adapted image communication system, *IEEE Trans. on Communications*, Vol. COM-32, pp. 1316-1322, 1984.

PETERSON (H.A.) *et al*: Quantization of colour image components in the DCT domain, *SPIE/IS&T 1991 Symposium on Electronic Imaging Science and Technology,* 1991.

M.4 Arithmetic coding references

LANGDON (G.): An Introduction to Arithmetic Coding, *IBM J. Res. Develop.*, Vol. 28, pp. 135-149, 1984.

PENNEBAKER (W.B.), MITCHELL (J.L.), LANGDON (G.) Jr., and ARPS (R.B.): An Overview of the Basic Principles of the Q-Coder Binary Arithmetic Coder, *IBM J. Res. Develop.*, Vol. 32, No. 6, pp. 717-726, 1988.

MITCHELL (J.L.) and PENNEBAKER (W.B.): Optimal Hardware and Software Arithmetic Coding Procedures for the Q-Coder Binary Arithmetic Coder, *IBM J. Res. Develop.*, Vol. 32, No. 6, pp. 727-736, 1988.

PENNEBAKER (W.B.) and MITCHELL (J.L.): Probability Estimation for the Q-Coder, *IBM J. Res. Develop.*, Vol. 32, No. 6, pp. 737-752, 1988.

MITCHELL (J.L.) and PENNEBAKER (W.B.): Software Implementations of the Q-Coder, *IBM J. Res. Develop.*, Vol. 32, No. 6, pp. 753-774, 1988.

ARPS (R.B.), TRUONG (T.K.), LU (D.J.), PASCO (R.C.) and FRIEDMAN (T.D.): A Multi-Purpose VLSI Chip for Adaptive Data Compression of Bilevel Images, *IBM J. Res. Develop.*, Vol. 32, No. 6, pp. 775-795, 1988.

ONO (F.), YOSHIDA (M.), KIMURA (T.) and KINO (S.): Subtraction-type Arithmetic Coding with MPS/LPS Conditional Exchange, *Annual Spring Conference of IECED*, Japan, D-288, 1990.

DUTTWEILER (D.) and CHAMZAS (C.): Probability Estimation in Arithmetic and Adaptive-Huffman Entropy Coders, submitted to *IEEE Trans. on Image Processing*.

JONES (C.B.): An Efficient Coding System for Long Source Sequences, *IEEE Trans. Inf. Theory,*Vol. IT-27, pp. 280-291, 1981.

LANGDON (G.): Method for Carry-over Control in a Fifo Arithmetic Code String, *IBM Technical Disclosure Bulletin*, Vol. 23, No.1, pp. 310-312, 1980.

M.5 Huffman coding references

HUFFMAN (D.A.): A Method for the Construction of Minimum Redundancy codes, *Proc. IRE,* Vol. 40, pp. 1098-1101, 1952.

Welcome to the JPEG Tutorial!

This page presents a brief description of how JPEG compresses images. JPEG, unlike other formats like PPM, PGM, and GIF, is a ***lossy*** compression technique; this means visual information is lost permanently. The key to making JPEG work is choosing what data to throw away.

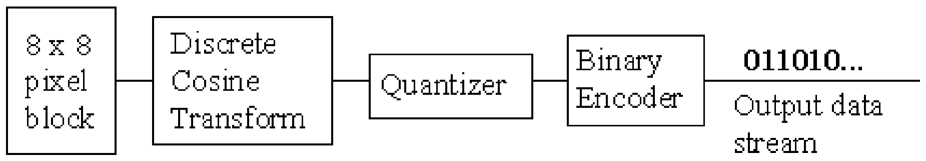
Introduction

JPEG is the image compression standard developed by the Joint Photographic Experts Group. It works best on natural images (scenes). This tutorial describes general JPEG compression for greyscale images; however, JPEG compresses color images just as easily. For instance, It compresses the red-green-blue parts of a color image as three separate greyscale images - each compressed to a different extent, if desired. The section, comparison of JPEG images, displays both color and black and white images compressed with JPEG.

How JPEG works

Figure one describes the JPEG process. JPEG divides up the image into 8 by 8 pixel blocks, and then calculates the discrete cosine transform (DCT) of each block. A quantizer rounds off the DCT coefficients according to the quantization matrix. This step produces the "lossy" nature of JPEG, but allows for large compression ratios. JPEG's compression technique uses a variable length code on these coefficients, and then writes the compressed data stream to an output file (*.jpg). For decompression, JPEG recovers the quantized DCT coefficients from the compressed data stream, takes the inverse transforms and displays the image. Figure 1 shows this process.

FIGURE 1. BLOCK DIAGRAM OF JPEG COMPRESSION



Testing Methods and Results

What does JPEG look like? How far can one compress an image and have it look identical to the original? presentable? merely recognizable? The comparison of JPEG images answers these questions. This section tests color and b/w pictures compressed at different levels. The original images are in PBM format. Version 3.0 of *XV, the imaging system for X-Windows,* implemented the JPEG compression.

References

Many books and articles discuss JPEG in more detail. An anonymous FTP site for more JPEG documentation is: ftp.uu.net/graphics/jpeg/.

Index of Hyperlinks

Below are all the hyperlinks used in this document.

Comments or Problems

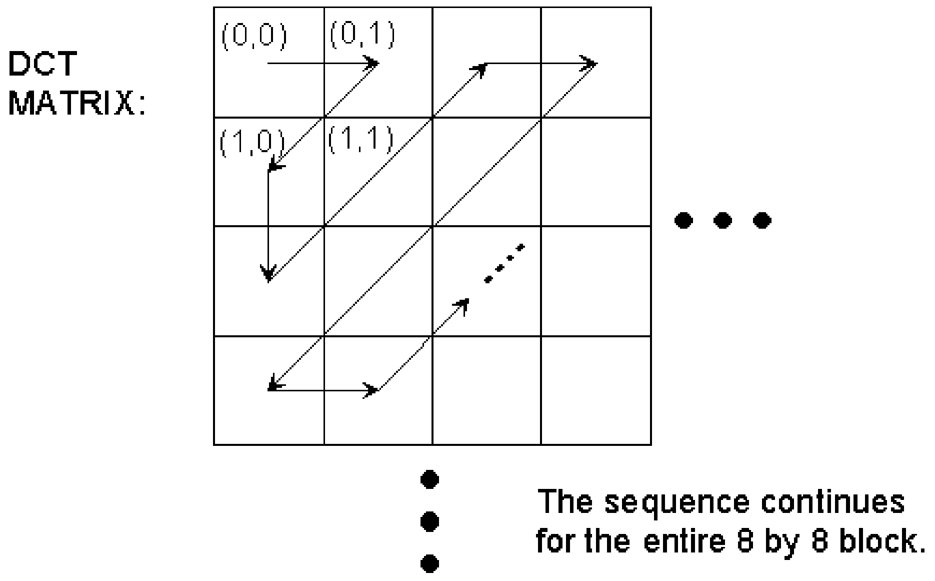
Thank you for reading this JPEG tutorial. If you have any problems with this document, or would like more information, please send email to ace@ecn.purdue.edu.

JPEG’s Compression Technique

Introduction

After quantization, it is not unusual for more than half of the DCT coefficients to equal zero. JPEG incorporates run-length coding to take advantage of this. For each non-zero DCT coefficient, JPEG records the number of zeros that preceded the number, the number of bits needed to represent the number’s amplitude, and the amplitude itself. To consolidate the runs of zeros, JPEG processes DCT coefficients in the zigzag pattern shown in figure two:

FIGURE 2. ZIG-ZAG SEQUENCE FOR BINARY ENCODING



Encoding

The number of previous zeros and the bits needed for the current number’s amplitude form a pair. Each pair has its own code word, assigned through a variable length code (for example Huffman, Shannon-Fano or Arithmetic coding). JPEG outputs the code word of the pair, and then the codeword for the coefficient’s amplitude (also from a variable length code). After each block, JPEG writes a unique end-of-block sequence to the output stream, and moves to the next block. When finished with all blocks, JPEG writes the end-of-file marker.

For Reference:

- The encoder’s Zig-Zag sequence.
- Top level of JPEG Tutorial

The Discrete Cosine Transform

Introduction

The discrete cosine transform (DCT) helps separate the image into parts (or spectral sub-bands) of differing importance (with respect to the image's visual quality). The DCT is similar to the discrete Fourier transform: it transforms a signal or image from the spatial domain to the frequency domain. With an input image, A, the coefficients for the output "image," B, are:

$$B(k_1, k_2) = \sum_{i=0}^{N_1-1} \sum_{j=0}^{N_2-1} 4 \cdot A(i,j) \cdot \cos\left[\frac{\pi \cdot k_1}{2 \cdot N_1} \cdot (2 \cdot i + 1)\right] \cdot \cos\left[\frac{\pi \cdot k_2}{2 \cdot N_2} \cdot (2 \cdot j + 1)\right]$$

The input image is N2 pixels wide by N1 pixels high; A(i,j) is the intensity of the pixel in row i and column j; B(k1,k2) is the DCT coefficient in row k1 and column k2 of the DCT matrix. All DCT multiplications are real. This lowers the number of required multiplications, as compared to the discrete Fourier transform. The DCT input is an 8 by 8 array of integers. This array contains each pixel's gray scale level; 8 bit pixels have levels from 0 to 255. The output array of DCT coefficients contains integers; these can range from -1024 to 1023. For most images, much of the signal energy lies at low frequencies; these appear in the upper left corner of the DCT. The lower right values represent higher frequencies, and are often small - small enough to be neglected with little visible distortion.

For Reference:

- Discrete cosine transform equation
- Top level of JPEG Tutorial

JPEG’s Quantizing Scheme

Introduction

There is a tradeoff between image quality and degree of quantization. A large quantization step size can produce unacceptably large image distortion. This effect is similar to quantizing Fourier series coefficients too coarsely; large distortion would result. Unfortunately, finer quantization leads to lower compression ratios. The question is how to quantize the DCT coefficients most efficiently. Because of human eyesight’s natural high frequency roll-off, these frequencies play a less important role than low frequencies. This lets JPEG use a much higher step size for the high frequency coefficients, with little noticeable image deterioration.

Quantization Matrix

The *quantization matrix* is the 8 by 8 matrix of step sizes (sometimes called *quantums*) - one element for each DCT coefficient. It is usually symmetric. Step sizes will be small in the upper left (low frequencies), and large in the upper right (high frequencies); a step size of 1 is the most precise. The quantizer divides the DCT coefficient by its corresponding quantum, then rounds to the nearest integer. Large quantums drive small coefficients down to zero. The result: many high frequency coefficients become zero, and therefore easier to code. The low frequency coefficients undergo only minor adjustment. The page explains how many zeros among the high frequency coefficients leads to efficient compression.

For Reference:

- Top level of JPEG Tutorial

Table 1. Compression Ratios for JPEG Quality Factors

Original size of color image (Kb)				313.076
Original size of B/W image (Kb)				104.437
Quality Factor	Color JPEG		B/W JPEG	
	File Size (Kb)	Comp. Ratio	File Size (Kb)	Comp. Ratio
75	23.039	13.59	21.02	4.97
20	8.457	37.02	7.599	13.74
5	4.009	78.09	3.257	32.07
3	3.268	95.80	2.522	41.41

POSTSCRIPT™
Software From Adobe

Encapsulated PostScript File Format Specification

Adobe Developer Support

Version 3.0

1 May 1992

Adobe Systems Incorporated

Corporate Headquarters
1585 Charleston Road PO Box 7900
Mountain View, CA 94039-7900
(415) 961-4400 Main Number
(415) 961-4111 Developer Support
Fax: (415) 961-3769

Adobe Systems Eastern Region
24 New England
Executive Park
Burlington, MA 01803
(617) 273-2120
Fax: (617) 273-2336

Adobe Systems Europe B.V.
Europlaza
Hoogoorddreef 54a
1101 BE Amsterdam Z-O, Netherlands
+31-20-6511 200
Fax: +31-20-6511 300

Adobe Systems Japan
Swiss Bank House 7F
4-1-8 Toranomon, Minato-ku
Tokyo 105, Japan
81-3-3437-8950
Fax: 81-3-3437-8968

PN LPS5002

Contents

Encapsulated PostScript File Format Specification

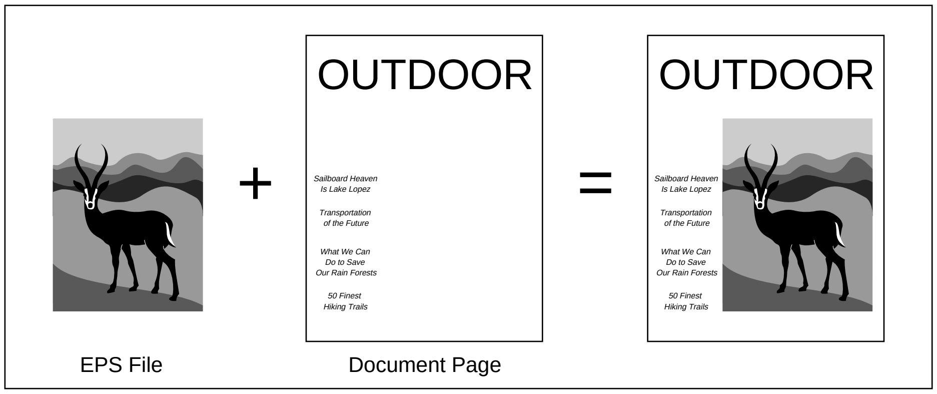
The encapsulated PostScript file (EPSF) format is a standard format for importing and exporting PostScript language files among applications in a variety of heterogeneous environments. This appendix details the format and contains specific information about the Macintosh® and MS-DOS® environments. The EPSF format is based on and conforms to the document structuring conventions (DSC) detailed in the *PostScript Document Structuring Conventions Specification* available from the Adobe Systems Developers Association. Proper use of the document structuring conventions is required when creating a PostScript language file that conforms to the EPSF format.

The main topics of this appendix include creating encapsulated PostScript (EPS) files, importing EPS files into other PostScript language files, and optional screen preview images for EPS files. Finally, a detailed example illustrates the concepts presented throughout this appendix.

1 Introduction

An encapsulated PostScript file is a PostScript language program describing the appearance of a single page. Typically, the purpose of the EPS file is to be included, or "encapsulated," in another PostScript language page description. The EPS file can contain any combination of text, graphics, and images, and it is the same as any other PostScript language page description with only a few restrictions. Figure 1 conceptually shows how an EPS file can be included in another PostScript language document.

Figure 1 *Document with an imported EPS file*


OUTDOOR
+
Sailboard Heaven Is Lake Lopez
Transportation of the Future
What We Can Do to Save Our Rain Forests
50 Finest Hiking Trails
=
OUTDOOR
Sailboard Heaven Is Lake Lopez
Transportation of the Future
What We Can Do to Save Our Rain Forests
50 Finest Hiking Trails
EPS File
Document Page

Applications that create conforming EPS files must follow the guidelines in section section 2.” There are two required DSC comments, some conditionally required comments, and several programming guidelines to ensure that the EPS file can be reliably imported into an arbitrary PostScript language page description without causing any side effects. An example of a side effect is erasing the page of the importing document or terminating the print job.

Applications that import EPS files must follow the guidelines in section section 3.” An application importing an EPS file must parse the EPS file for DSC comments and extract at least the bounding box and resource dependencies of the EPS file. The application should also read and display the screen preview, if present. If there is no screen preview provided in the EPS file, the application must provide an alternate representation and allow the user to place and transform the preview on the screen.
The application must then convert the user’s manipulations into the appropriate transformation to the PostScript coordinate system before sending the document to the printer. The application must also preserve its stacks, current dictionary, and graphics state before the imported EPS file is executed.

Note that EPS files are a *final-form* representation. They cannot be edited when imported into a document. However, the imported EPS file as a whole may be manipulated to some extent, including transformations such as translation, rotation, scaling, and clipping.

The device-independent nature of the PostScript language makes it an excellent interchange format. However, it normally requires a PostScript language interpreter to preview an EPS file on screen. Display PostScript systems allow EPS files to be dynamically interpreted, insuring the highest-quality, on-screen preview regardless of scale, rotation, or monitor type. For other environments where the Display PostScript system is not available, the EPS file format allows for an optional screen preview image.

The format of this preview representation varies from system to system. It is typically a Macintosh PICT resource, a TIFF file, or a device-independent hex bitmap. If the EPS file does not provide a preview image, the application that includes the EPS file must provide a representation of the preview, such as a gray box that represents the extent of the EPS file. The end user can use the screen preview to position and size the EPS file in the document.

To support encapsulated PostScript files effectively, some cooperation is required among the applications that *produce* EPS files and those that *use* EPS files. Typically, EPS files are used by importing (or including) them in other documents.

All DSC comments in an EPS file communicate information. How an application uses this information is up to the programmer of the including application. When importing an EPS file, do not reduce the amount of information in the EPS file by improperly removing or altering DSC comments. In general, the comments indicate what resources and language extensions are used, and where they are used in the EPS file. Encapsulated PostScript files are final-form print files that do not know anything about the printer on which they will be imaged. If they have specific resource needs, such as fonts, these needs must be carefully preserved and addressed.

Any application that generates PostScript language programs is potentially both a *consumer* and a *producer* of encapsulated PostScript files. It is probably best not to think that an application is at either end of the chain. If an application imports an EPS file, it is responsible for reading and understanding any of the resource needs of the imported EPS file. These needs must be reflected in the resource usage comments of the composite document the including application creates. For example, if an imported EPS file uses Lithos™, but the rest of the document is set in Times-Roman, then by importing the EPS file, the document now also uses the Lithos font. This fact must be reflected in the composite document's outermost %%DocumentNeededFonts: comment. This concept holds true for the %%DocumentNeededResources:, %%LanguageLevel: and %%Extensions: comments as well.

2 Guidelines for Creating EPS Files

To be considered a conforming EPSF version 3.0 file, a file must follow the rules set forth in this appendix, be a *single* page document that fully conforms to the DSC version 3.0 or later (described in the *PostScript Document Structuring Conventions Specifications* available from the Adobe Systems Developers' Association), and include two required DSC header comments.

2.1 Required DSC Header Comments

The two required DSC Header comments are

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: llx lly urx ury
```

The first required DSC header comment informs the including application that the file conforms to version 3.0 of the EPSF format as described in this appendix. This is the version comment.

The second required DSC header comment provides information about the size of the EPS file and must be present so the including application can transform and clip the EPS file properly. This is the bounding box comment.

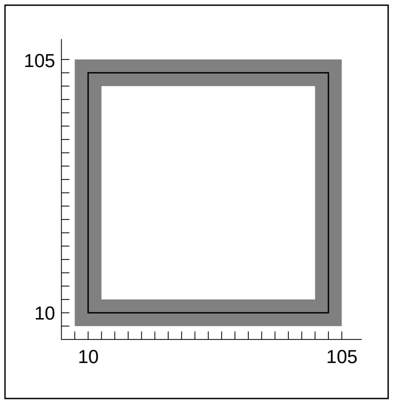
The four arguments of the bounding box comment correspond to the lower-left (*llx, lly*) and upper-right (*urx, ury*) corners of the bounding box. They are expressed in the default PostScript coordinate system. For an EPS file, the bounding box is the smallest rectangle that encloses all the marks painted on the *single* page of the EPS file. Graphics state information, such as the current line width and line join parameters, must be considered when calculating the bounding box. Example 1: shows a minimally conforming EPS file that draws a square with a line width of 10 units.

Example 1:

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 5 5 105 105
10 setlinewidth
10 10 moveto
0 90 rlineto 90 0 rlineto 0 -90 rlineto closepath
stroke
```

The marks painted by Example 1:, and how they are positioned with respect to the PostScript coordinate system, are illustrated in Figure 2. If the line width were not considered when calculating the bounding box, the bounding box would be incorrectly positioned by five units on each side of the square, causing the application to incorrectly place and clip the imported EPS file. The bounding box specified for this example is correct.

Figure 2 *Calculating the correct bounding box*



Regardless of the coordinate system in which an application operates, there is a convenient way to estimate the bounding box: Print the page, then use a point ruler to measure from the lower-left corner of the paper to the lower-left corner of the image. Then measure to the upper-right corner, also using the lower-left corner of the paper as the origin. These two measurements give the bounding box and do not depend on any computation.

2.2 Conditionally Required Comments

There are several optional DSC comments that may be conditionally required for a conforming EPS file. These comments must appear in an EPS file if certain features are present—for example, comments to bracket the preview section or to state that a certain language version or language extensions must be present in the interpreter.

The %%Begin(End)Preview comments must bracket the preview section of an EPS file if the preview is represented in the encapsulated PostScript interchange (EPSI) format. See section section 6," for details and an example of EPSI.

The %%Extensions: comment is required if the EPS file requires a PostScript language interpreter that supports particular PostScript language extensions to print properly. For example, the EPS file may contain CMYK language extension operators and must be sent to a printer that can handle those operators. In such a case, the EPS file must contain either the %%Extensions: CMYK or the %%LanguageLevel: 2 comment.

The %%LanguageLevel: comment is required if the EPS file uses Level 2 features without providing conditional emulation. With this information, the including application can alert the user and avoid any errors that would be generated if the file were sent to a Level 1 printer.

If the EPS file uses language extensions or Level 2 features, and it provides complete emulation of the features in terms of Level 1 operators, the %%Extensions: and %%LanguageLevel: comments are not necessary. See Appendix D of the *PostScript Language Reference Manual, Second Edition* for compatibility and emulation strategies.

If the EPS file requires any fonts, files, forms, patterns, procsets (procedure sets), or any other resources, the appropriate DSC comment must appear in the header comments section of the file. See the *PostScript Document Structuring Conventions Specifications* available from the Adobe Systems Developers' Association.

2.3 Recommended Comments

An application or spooler may optionally use the general header comments %%Creator:, %%Title:, and %%CreationDate: to provide information about a document. These header comments are strongly recommended for EPS files.

2.4 Illegal and Restricted Operators

There are some PostScript language operators plus **statusdict** and **userdict** operators that are intended for system-level jobs or page descriptions that are not appropriate in an EPS file. In addition to all operators in **statusdict** and the operators in **userdict** for establishing an imageable area, the following operators must not be used in an EPS file:

banddevice	exitserver	initmatrix	setshared
clear	framedevice	quit	startjob
cleardictstack	grestoreall	renderbands	
copypage	initclip	setglobal	
erasepage	initgraphics	setpagedevice	

If used properly, the following operators are allowed in an EPS file. However, use of any of these must comply with the rules in Appendix I of the *PostScript Language Reference Manual, Second Edition*. Improper use can cause unpredictable results.

nulldevice	sethalftone	setscreen	undefinefont
setgstate	setmatrix	settransfer	

2.5 Stacks and Dictionaries

The PostScript interpreter's operand and dictionary stacks *must* be left in the state they were in before the EPS file was executed. The EPS file must not leave objects on either of these two stacks as a result of its execution. All operators placed on the operand stack must be used or removed from the stack with the **pop** operator.

It is strongly recommended that an EPS file make all of it definitions in its own dictionary. This means an EPS file should create its own dictionary or dictionaries instead of writing into the importing application's dictionary. In Level 1 interpreters, the dictionary the importing application uses may not have room for the EPS file definitions. Also, to avoid the possibility of an **invalidrestore** error, make sure the EPS file's dictionary is removed from the dictionary stack using the PostScript language operator **end** when the EPS file has finished using it. Every dictionary that the EPS file places on the dictionary stack with a **begin** operator must be removed from the dictionary stack by the EPS file with a corresponding **end** operator.

Note *Do not use the* **clear** *or* **cleardictstack** *operators to clear the stacks in an EPS file. These wholesale cleanup operators not only clear the EPS file's operands and dictionaries from the stacks, they may clear other objects as well.*

The PostScript dictionary lookup mechanism searches the dictionaries that are on the dictionary stack. Bypassing the dictionary lookup mechanism for system-level names is *illegal* in an EPS file. *Do not use the following type of code:*

```
/S systemdict /showpage get def% Illegal EPS code
```

It may cause incorrect results in the including application's PostScript output by overriding the application's redefinitions.

2.6 Graphics State

An application importing an EPS file may transform the PostScript coordinate system or alter some other aspect of the graphics state so it is no longer in its default state. This allows the application to change the appearance of the EPS file, typically by resizing, clipping, or rotating the illustration. If the EPS file makes assumptions about the graphics state, such as the current clipping path, or explicitly sets something it shouldn't, such as the transformation matrix (see section section 2.4”), the results may not be what were expected.

In preparation for including an EPS file, the graphics state must be set by the including application as follows: current color to black, line caps to square butt end, line joins to mitered, line width to 1, dash pattern to solid, miter limit to 10, and current path to an empty path. Also, if printing to a Level 2 interpreter, overprint and stroke adjust should be set to *false*. An EPS file can assume that this is the default state. It is the responsibility of the application importing the EPS file to make sure that the graphics state is correctly set.

2.7 Initializing Variables

It is common for PostScript language programs to use short names, such as `x`, for variables or procedures. Name-conflict problems can occur if an EPS file does not initialize its variables *before* defining its procedures—in particular, before binding them. In the following example, the variable `x` is not initialized before being used in the procedure `proc1`. Because the value of `x` in the enclosing program happens to be an operator, **bind** causes the name `x` to be replaced by the operator **lineto** in `proc1`. This causes a **stackunderflow** error upon execution.

```
%!PS-Adobe-3.0
...Document prolog of including application...
/x /lineto load def      % Application defines x to be lineto
```

```
...More of document prolog and setup...
%%BeginDocument: GRAPHIC.EPS
...Document prolog and setup for EPS file...
/proc1 {                 % Enter deferred execution mode
  /x exch def
  x 4 moveto
  } bind def             % x associated with lineto after bind
4 proc1                  % Execute proc1 and cause error
...Rest of EPS file...
%%EndDocument
...Rest of including application document...
```

In the following example, the EPS file *correctly* initializes the variable x before defining the procedure proc1:

```
%!PS-Adobe-3.0
...Document prolog of including application...
/x /lineto load def      % Application defines x to be lineto
...More of document prolog and setup...
%%BeginDocument: GRAPHIC.EPS
...Document prolog and setup for EPS file...
/x 0 def                 % Initialize variables before defining procs
/proc1 {
  /x exch def
  x 4 moveto
} bind def
4 proc1                  % Execute Proc1
...Rest of EPS file...
%%EndDocument
...Rest of including application document...
```

2.8 Ensuring Portability

Although using outside resources, such as fonts, patterns, files, and procsets, is allowed in an EPS file, the most portable files are those that are self-contained and do not rely on outside resources. For example, if an EPS file requires an encoding other than the default encoding for a font, then the EPS file should perform the re-encoding.

EPS files must never rely on procedures that are defined in application- or driver-provided prologs, such as procedures defined in the Apple LaserPrep file. Such definitions might or might not be present, depending on the actions of the enclosing program or previous jobs.

Because EPS files should be portable across heterogenous environments, 7-bit ASCII is the recommended format for data in EPS files. Although binary data is allowed, use caution when producing data that is expected to be portable. The use of binary data may make it impossible to print on some printers across some communication channels. Binary data that has special meaning, such as “flow control” or “marking the end of a file,” can cause file transmission problems in certain communications environments. For example, the control-D character is used as an end-of-file indicator in serial and parallel communications channels. Because this character terminates the job in serial and parallel environments, it is not prudent to produce an EPS file with this character in it.

See Appendix D of the *PostScript Language Reference Manual, Second Edition* for guidelines about how to take advantage of language extensions and Level 2 features while maintaining compatibility with Level 1 PostScript interpreters.

2.9 Miscellaneous Constraints

EPS files must not have lines of ASCII text that exceed 255 characters, excluding line-termination characters.

Lines must be terminated with one of the following combinations of characters: CR, LF, CR LF, or LF CR.

CR is the carriage return character and LF is the line feed character (decimal ASCII 13 and 10, respectively).

3 Guidelines for Importing EPS Files

This section contains guidelines that should be followed when creating an application that imports EPS files. The first part discusses displaying an EPS file; the second covers producing the PostScript language code for the printer.

This section contains several PostScript language code fragments. A complete code example that implements all of these segments is in section section 7.”

3.1 Displaying an EPS File

There are several techniques for including an EPS file in a document. The following scenario is typical:

1. When the user imports an EPS file, the application prompts the user to select the EPS file to be imported.

2. The application opens the selected file and parses it for useful information. If either of the two required header comments is missing, the application should alert the user that the file is not a conforming EPS file and abort the import.

The DSC elementary type (atend) may be used to defer bounding box data to the end of the EPS file. This means an application may need to parse through the %%Trailer comments to obtain the bounding box data.

3. If the version and bounding box comments are found, the application should prompt the user to place the EPS file. It should then display the screen preview. If no preview is provided with the EPS file, the application must provide a representation of the EPS file.

   If the application must create its own representation, a gray box matching the extent of the bounding box with some information in it suffices. The information should at least include the title of the EPS file. This can be obtained from the DSC header comment: %%Title:. Other information, such as %%Creator: and %%Creation-Date:, may also be displayed.

The bounding box comment can be used to help determine scaling factors and the proportions of the illustration. The including application should enable the user to specify a "placement box" to display the screen preview or the application-supplied representation of the screen preview if there is not a preview present in the EPS file.

The bounding box can be used to calculate a ratio that the application can use if the user wants to maintain original proportions while specifying a placement box. Alternately, the application may display the preview full size, and then allow the user to size and place the graphic as desired. Regardless of the method used to display the preview initially, the user should have the option of maintaining the original proportions supplied by the bounding box or distorting the proportions of the EPS graphic.

3.2 Producing a Composite PostScript Language Program

The following guidelines must be considered when producing a composite PostScript language program that includes an imported EPS file.

Use save and restore

An application should encapsulate the imported EPS file in a **save**/**restore** construct. This allows all VM the EPS file uses to be recovered and the graphics state to be restored.

Redefine showpage

The **showpage** operator is permitted in EPS files because it is present in so many PostScript language files. Therefore, it is reasonable for an EPS file to use the **showpage** operator, although it is not necessary if the EPS file will only be imported into another document. The application importing the EPS file is responsible for redefining **showpage**. **showpage** may be redefined using the following code segment:

```
/showpage { } def
```

Prepare the Graphics State

In preparation for including an EPS file, the including application must set the graphics state as follows: current color to black, line caps to square butt end, line joins to mitered, line width to 1, dash pattern to solid, miter limit to 10, and the current path should be set to an empty path. This state can be explicitly set using the following code segment:

```
0 setgray 0 setlinecap 1 setlinewidth
0 setlinejoin 10 setmiterlimit [ ] 0 setdash newpath
```

Also, if printing directly to a Level 2 printer, the overprint and stroke adjust graphics state parameters must be set to *false*. This can be done by conditionally using the following code segment:

```
false setoverprint false setstrokeadjust
```

Note *If the application knows that any given parameter of the current graphics state is already in its default state, there is no need to execute the related PostScript language code to reset that parameter.*

Push userdict

It is recommended that an application importing an EPS file use the **begin** operator to push a copy of **userdict** on top of the dictionary stack. Ideally, the imported EPS file should create its own dictionary, but if it does not, and if the application's dictionary does not have enough room for the EPS file's definitions**,** a **dictfull** error may result when the EPS file makes its definitions. After execution of the EPS file, the application should remove the copy of **userdict** from the dictionary stack by executing the **end** operator.

Clear the Operand Stack

The application importing the EPS file must leave an empty operand stack for the EPS file. It is reasonable for the EPS file to expect that the entire operand stack be available for its own use. If the entire operand stack is needed and is not available, a **stackoverflow** error may occur. Also, if the operand stack is empty, an EPS file that inappropriately executes **clear** will not cause any problems.

Protect the Stacks

An EPS file should leave the operand and dictionary stacks as they were before the EPS file was executed. However, this may not always be the case. So before including the EPS file, the importing application should be sure to count the number of objects on the dictionary and operand stacks.

Then, after executing the EPS file, it should make sure the stacks contain the same number of objects as they did before the EPS file was executed. The following code segment shows how to obtain the count of objects on the dictionary and operand stacks:

```
/Dict_Count countdictstack def
/Op_Count count def
```

Bracket EPS File with Comments

The included EPS file must be bracketed by the %%Begin(End)Document: comments as described in the *PostScript Document Structuring Conventions Specifications* available from the Adobe Systems Devlopers' Association.

Handle Special Requirements

If either the %%LanguageLevel: comment or the %%Extensions: comment is present in the header comments section of the EPS file, then at print time the application printing the composite file is responsible for assuring that the printer can handle the specified language extensions. If the application determines that the printer does not have the necessary language features to print the document properly, or if the application cannot determine extension availability, the user should be notified and prompted for the appropriate action. Also, if an application has imported an EPS file that requires extensions, the application's output is now dependent on the *same* extensions. This must be reflected in the document's header comment section.

If any %%DocumentNeededResources: or %%DocumentNeededFonts: comments are present in the header comments section of the EPS file, before printing the document the application must be sure the resources are available. If any of the resource requirements cannot be handled, the user must be notified and prompted for an appropriate action. Such an action may involve having the user locate the resource or allowing the user or document manager to reroute the print job to a printer that has the required resources. Also, if an application has included an EPS file that requires these comments, the application's output is now dependent on the same resources. This must be reflected in the document's header comment section.

Default Coordinate System Transformation

Before including the EPS file in its page description, the importing application must transform the PostScript coordinate system according to the final user placement of the EPS file. The order of the transformation sequence must be:

1. Translate the origin to the new user-chosen origin.

2. Rotate, if the user has rotated the EPS file.

3. Scale, if the user has changed the size.

4. Translate the lower-left corner of the EPS file's bounding box to the user-chosen origin.

Details on transforming the PostScript coordinate system are below. The first example is a simple case in which the user coordinate system matches the default PostScript coordinate system. The second example is a general case transformation from application space to the default PostScript coordinate system.

Figure 3 shows an EPS file and its bounding box superimposed on a target page. The EPS file is shown as it would be drawn if the EPS file were printed without first transforming the PostScript coordinate system. The placement box in the upper-right corner of the page shows where the user chose to place the EPS file.

Figure 3 *EPS file and placement box*

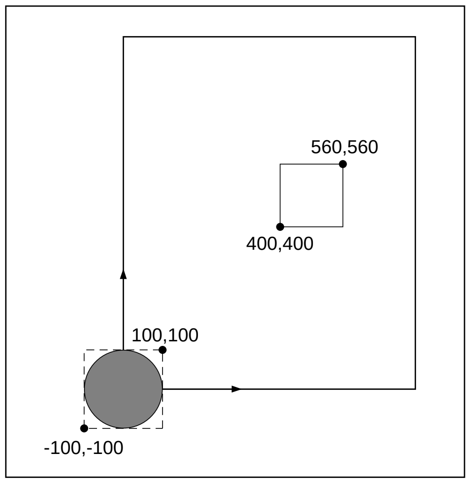
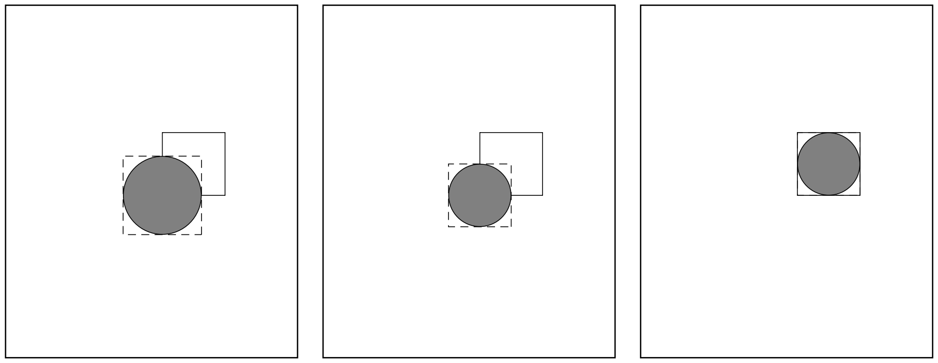


Figure 4 contains three diagrams that show the steps necessary to properly translate and scale the PostScript coordinate system to achieve the user-chosen placement on the page.

Figure 4 *Transforming the EPS file*



Translate to new origin *Scale to fit placement box* *Translate to final position*

Assuming that the bounding box found in the header of the EPS file is %%BoundingBox: -100 -100 100 100, the following PostScript language code fragment properly places the EPS file on the printed page:

```
400 400 translate        % Translate to new origin
.8 .8 scale              % Scale to fit “placement
box”
100 100 translate        % –llx –lly translate
```

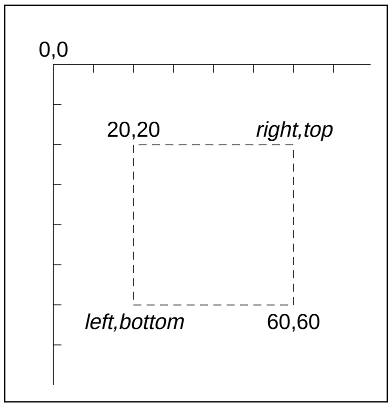
This transformation code must be inserted into the PostScript stream *ahead* of the EPS code being sent to the printer.

Figures H.3 and H.4 and the corresponding PostScript code fragment assume that the application coordinate system matches the default PostScript coordinate system. The following section discusses a more general coordinate system transformation.

General Coordinate System Transformation

Typically, an application transforms the PostScript coordinate system so the native drawing units of the application space can be used as the operands to the PostScript language operators defining the page. Consider Figure 5, which represents an arbitrary application coordinate system and a placement box for an EPS file.

Figure 5 *Application coordinate system plus placement box*



To transform the PostScript coordinate system to match the application coordinate system in Figure 5, an application could execute the following code fragment:

```
0 792 translate
1 –1 scale
```

This assumes that each unit of application space is equal to one PostScript unit. If one unit in application space were equal to five PostScript units, then the transformation might look like this:

```
0 792 translate
5 –5 scale
```

Assuming that the coordinate system has already been properly translated and scaled from the PostScript coordinate system to the application coordinate system as above, then the following steps can be used to place the EPS file in the user-chosen box:

1. *left bottom* **translate**

2. ((*right* – *left*)/(*urx* – *llx*)) (*top* – *bottom*)/(*ury* – *lly*) **scale**

3. – (*llx*) – (*lly*) **translate**

where *bottom*, *left*, *top*, and *right* are coordinates of the placement box in application space, and *llx*, *lly*, *urx*, and *ury* are bounding box parameters the EPS file supplies.

As a final example, assume that the PostScript coordinate system has already been transformed to match the application coordinate system, the EPS file bounding box is %%BoundingBox: 20 20 100 100**,** and the user-chosen placement box is the box shown in Figure 5 on page 15. Using the formula and steps above, the transformation before executing the included EPS file would be as follows:

```
20 60 translate
.5 –.5 scale
–20 –20 translate
```

Set Up a Clipping Path

The importing application should set up a clipping path around the imported EPS file. This can be accomplished by setting a clipping path that corresponds to the bounding box of the imported EPS file after making the PostScript coordinate system transformations or by allowing the user to optionally supply an arbitrary clipping path for special effects.

Discard the Screen Preview

If an EPS file includes a screen preview in EPSI format, the importing application should discard the preview before sending the document to a printer. Although the EPSI preview is represented by PostScript comments and will not pose a problem when included in the PostScript language file sent to the printer, it takes extra time to transmit the preview.

If the preview in the EPS file is in Macintosh PICT format, do not include the PICT resource in the PostScript language file sent to the printer.

If the preview is in TIFF format or in Microsoft® Windows™ Metafile format, take care to extract the PostScript language code that is to be sent to the printer. See section section 5.2,” for details.

If the EPS file does not include a screen preview, the entire EPS file can be included in the PostScript language file sent to the printer.

Maintain EPSF Version 2.0 Compatibility

The EPSF version 3.0 requires that an EPS file leave the operand and dictionary stacks as they were before the EPSF was executed. However, this was not explicitly stated in earlier versions of the EPSF format. Therefore, before including the EPS file, be sure to count the number of objects on the dictionary and operand stacks. After executing the EPS file, make sure the stacks contain the same number of objects they did before the EPS file was executed.

Preparation for Including an EPS File

Example 2: shows procedure BeginEPSF, which an application might use to prepare to include an EPS file in its print stream. Execute the BeginEPSF procedure before the EPS file.

Example 2:

```
/BeginEPSF { %def
  /b4_Inc_state save def           % Save state for cleanup
  /dict_count countdictstack def   % Count objects on dict stack
  /op_count count 1 sub def        % Count objects on operand stack
  userdict begin                   % Push userdict on dict stack
  /showpage { } def                % Redefine showpage, { } = null proc
  0 setgray 0 setlinecap           % Prepare graphics state
  1 setlinewidth 0 setlinejoin
  10 setmiterlimit [ ] 0 setdash newpath
  /languagelevel where             % If level not equal to 1 then
  {pop languagelevel               % set strokeadjust and
  1 ne                             % overprint to their defaults.
    {false setstrokeadjust false setoverprint
    } if
  } if
} bind def
```

Example 3: shows procedure EndEPSF, which illustrates how to restore the PostScript state to the way it was before inclusion and execution of the EPS file. Execute the EndEPSF procedure after the EPS file.

Example 3:

```
/EndEPSF { %def
  count op_count sub {pop} repeat       % Clean up stacks
  countdictstack dict_count sub {end} repeat
  b4_Inc_state restore
} bind def
```

Example 4: illustrates use of the BeginEPSF and EndEPSF procedures.

Example 4:

```
BeginEPSF                               % Prepare for the included EPS file
left bottom translate                   % Place the EPS file
angle rotate
Xscale Yscale scale
-llx -lly translate
...Set up a clipping path...
%%BeginDocument: MyEPSFile
...Included EPS file here...
%%EndDocument
EndEPSF                                 % Restore state, and cleanup stacks
```

4 File Types and Naming

EPS files have become a standard format for importing and exporting PostScript language files among applications in a variety of heterogenous environments. This section contains specific information about file types and naming conventions in a variety of environments.

4.1 Apple Macintosh File System

The Macintosh file type for application-created PostScript language files is EPSF. Files of type TEXT are also allowed so users can create EPS files with standard text editors. However, the DSC must still be strictly followed. A file of type EPSF should contain a PICT resource in the resource fork of the file containing a screen preview image of the EPS file. The file name may follow any naming convention as long as the file type is EPSF. If the file type is TEXT, the extensions .epsf, and .epsi should be used for EPS files with Macintosh-specific and device-independent preview images, respectively. See sections section 5," and section 6."

4.2 MS-DOS and PC-DOS File System

The recommended file extension is .EPS. For EPS files that provide an EPSI preview, the recommended extension is .EPI. Because the name and extension may be user-supplied, it is recommended that the application provide a default extension of .EPS or, if the file includes an EPSI preview, the application can provide .EPI as the default extension.

4.3 Other File Systems

Although naming is file-system dependent, in general the extension .epsf is the preferred way to name an EPS file. Likewise, .epsi is the preferred extension for the interchange format. In systems where lower-case letters are not recognized or are not significant, all upper-case letters can be used.

5 Device-Specific Screen Preview

The EPS file usually has a graphic screen preview so it can be transformed and displayed on a computer screen to aid in page composition before printing. Depending on the capabilities of the importing application, the user may position, scale, clip, or rotate this screen representation of the EPS file. The composing software should keep track of these transformations and reflect them in the PostScript language code that is ultimately sent to the printer.

The exact format of this screen representation is machine-specific. That is, each computing environment may have its own preferred preview image format, which is typically the appropriate screen representation for that environment. Also, a device-independent screen representation called EPSI is specified in section section 6.” It is recommended that all applications support this format.

5.1 Apple Macintosh PICT Resource

A QuickDraw™ representation of the EPS file can be created and stored as a PICT resource in the resource fork of the EPS file. It must be given resource number 256. If the PICT exists, the importing application may use it for screen display. If the *picframe* is transformed to PostScript language coordinates, it should agree with the %%BoundingBox: comment.

Given the size limitations on PICT images, the *picframe* and bounding box may not always agree. If there is a discrepancy, the %%BoundingBox: must always be taken as the “truth,” because it accurately describes the area the EPS file will image.

5.2 Windows Metafile or TIFF

Either a Microsoft Windows Metafile or a TIFF (tag image file format) section can be included as the screen representation of an EPS file.

The EPS file has a binary header added to the beginning that provides a sort of table of contents to the file. This is necessary because there is not a second “fork” in the file system as there is in the Macintosh file system.

Note *It is always permissible to have a pure ASCII PostScript language file as an EPS file in the DOS environment.*

The importing application must check the first 4 bytes of the EPS file. If they match the header as shown in Table 1, the binary header should be expected. If the first two match %!, it should be taken to be an ASCII PostScript language file.

Table 1 *DOS EPS Binary File Header*

Bytes	*Description*
0-3	Must be hex C5D0D3C6 (byte 0=C5).
4-7	Byte position in file for start of PostScript language code section.
8-11	Byte length of PostScript language section.
12-15	Byte position in file for start of Metafile screen representation.
16-19	Byte length of Metafile section (*PSize*).
20-23	Byte position of TIFF representation.
24-27	Byte length of TIFF section.

28-29	Checksum of header (XOR of bytes 0-27). If Checksum is FFFF then ignore it.

It is assumed that either the Metafile or the TIFF position and length fields are zero. That is, only one or the other of these two formats is included in the EPS file.

The Metafile must follow the guidelines the Windows specification sets forth. It should not set the *viewport* or *mapping mode*, and it should set the *window origin* and *extent*. The application including the EPS file should scale the picture to fit within the %%BoundingBox: comment specified in the EPS file.

6 Device-Independent Screen Preview

This screen preview format is designed to allow EPS files to be used as an interchange format among widely varied systems. The preview section of the file is a bitmap represented as ASCII hexadecimal to be simple and easily transportable. This format is called encapsulated PostScript interchange format, or EPSI.

An EPSI file is truly portable and requires no special code for decompressing or otherwise understanding the bitmap portion, other than the ability to understand hexadecimal notation.

The %%BeginPreview: *width height depth lines* and %%EndPreview comments bracket the preview section of an EPSI file. The *width* and *height* fields provide the number of image samples (pixels) for the preview. The *depth* field provides the number of bits of data used to establish one sample pixel of the preview—typical values are 1, 2, 4, 8. An image that is 100 pixels wide will always have 100 in the *width* field, although the number of bytes of hexadecimal needed to build that line will vary if *depth* varies. The *lines* field tells how many lines of hexadecimal are contained in the preview, so an application that does not care may easily skip them. All arguments are integers.

The bit order of the preview image data is the same as the bit order used by the **image** operator. That is, the preview image is considered to exist in its own coordinate system. The rectangular boundary of the preview image has its lower-left corner at (0,0) and its upper-right corner at (*width*, *height*). The byte order is fixed and should be (0,0) through (*width* – 1), then (0,1) through (*width* – 1,1), etc.

6.1 Guidelines for EPSI Files

The following guidelines are to clarify a few basic assumptions about the EPSI format, which is intended to be extremely simple because its purpose is for interchange. No system should have to do much work to decipher EPSI files. The format is accordingly kept simple and option free.

- The preview section must appear after the header comment section, but before the document prologue definitions. That is, it should immediately follow the %%EndComments: line in the EPS file.

- In the preview section, 0 is white and 1 is black. Arbitrary transfer functions and “flipping” black and white are not supported. Note that in the PostScript language, 0 and 1 have the opposite meaning (0 is black and 1 is white) for the **setgray** operator.

- The preview image can be of any resolution. The size of the image is determined solely by its bounding box, and the preview data should be scaled to fit that rectangle. Thus, the *width* and *height* parameters from the image are *not* its measured dimensions, but rather describe the amount of data supplied for the preview. Only the bounding rectangle describes the dimensions.

- The hexadecimal lines must never exceed 255 bytes in length. In cases where the preview is very wide, the lines must be broken. The line breaks can be made at any even number of hex digits, because the dimensions of the finished preview are established by the *width*, *height*, and *depth* values.

- All non-hexadecimal characters must be ignored when collecting the data for the preview, including tabs, spaces, newlines, percent characters, and other stray ASCII characters. This is analogous to the **readhexstring** operator.

- Each line of hexadecimal begins with a percent character (%). This makes the entire preview section a PostScript language comment to be ignored by the PostScript interpreter. The file can be printed without modification.

- Although the EPSI hex preview can be sent to the printer, to shorten transmission time it is recommended that the preview image be stripped out of the document before transmitting the file to the printer.

- The data for each scan line of the image must be a multiple of 8 bits long. If necessary, pad the end of the scan line data with 0’s.

Example 5: is a sample EPSI format file. Remember there are 8 bits to a byte, and that it requires 2 hexadecimal digits to represent one binary byte. Therefore, the 80-pixel width of the image requires 20 bytes of hexadecimal data, which is (80 / 8) **x** 2. The PostScript language segment simply draws a box, as can be seen in the last few lines.

Example 5:

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 80 24
%%Pages: 0
%%Creator: John Smith
%%CreationDate: November 9, 1990
%%EndComments
%%BeginPreview: 80 24 1 24
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%FF0000000000000000FF
%FF0000000000000000FF
%FF0000000000000000FF
%FF0000000000000000FF
%FF0000000000000000FF
%FF0000000000000000FF
%FF0000000000000000FF
%FF0000000000000000FF
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%FFFFFFFFFFFFFFFFFFFF
%%EndPreview
%%EndProlog
%%Page: "one" 1
4 4 moveto 72 0 rlineto 0 16 rlineto -72 0 rlineto
closepath
8 setlinewidth stroke
%%EOF
```

7 EPS Example

The following example illustrates the proper use of DSC comments in a typical page description that an application might produce when including an EPS file. For an EPS file that is represented as

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 4 4 608 407
%%Title: (ARTWORK.EPS)
%%CreationDate: (10/17/89) (5:04 PM)
%%EndComments
...PostScript code for illustration..
showpage
%%EOF
```

the including document's page description, including the imported EPS file, would be represented as

```
%!PS-Adobe-3.0
%%BoundingBox: 0 0 612 792
%%Creator: SomeApplication
%%Title: (Smith.Text)
%%CreationDate:  11/9/89 (19:58)
%%Pages: 1
%%DocumentFonts: Times-Roman Times-Italic
%%DocumentNeededFonts: Times-Roman Times-Italic
%%EndComments

%%BeginProlog
/ms {moveto show} bind def
/s /show load def
/SF { %/FontIndex FontSize /FontName SF --
  findfont exch scalefont dup setfont def
} bind def
/sf /setfont load def
/rect { % llx lly w h  % Used to create a clipping path
  4 2 roll moveto
  1 index 0 rlineto
  0 exch rlineto
  neg 0 rlineto
  closepath
} bind def


/BeginEPSF { %def           % Prepare for EPS file
  /b4_Inc_state save def% Save state for cleanup
  /dict_count countdictstack def
  /op_count count 1 sub def % Count objects on op stack
  userdict begin         % Make userdict current dict
  /showpage { } def      % Redefine showpage to be
null
  0 setgray 0 setlinecap
  1 setlinewidth 0 setlinejoin
  10 setmiterlimit [ ] 0 setdash newpath
  /languagelevel where   % If level not equal to 1 then
    {pop languagelevel     % set strokeadjust and
    1 ne                 % overprint to their defaults
    {false setstrokeadjust false setoverprint
    } if
  } if
}bind def
/EndEPSF { %def
  count op_count sub {pop} repeat
% Clean up dict stack
  countdictstack dict_count sub {end} repeat
  b4_Inc_state restore
} bind def
%%EndProlog

%%BeginSetup
%%IncludeFont: Times-Roman
%%IncludeFont: Times-Italic
%%EndSetup
%%Page: 1 1
%%BeginPageSetup
/pgsave save def
%%EndPageSetup
/F1 40 /Times-Roman  SF
```
...Set some text with F1...
```
/F2 40 /Times-Italic SF
```
...Set some text with F2...
```
F1 sf
```
...Set some more text with F1...
```
F2 sf
```
...Set some more text with F2...
```
BeginEPSF
65.2 10 translate          % Position the EPS file
.80 .80 scale              % Scale to desired size
-4 -4 translate            % Move to lower left of the
EPS
4 4 604 403 rect           % Set up clipping path
clip newpath               % Set the clipping path


%%BeginDocument: ARTWORK.EPS
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox:  4 4  608 407
%%Title:  (ARTWORK.EPS)
%%CreationDate:  (10/17/90) (5:04 PM)
%%EndComments
```
...PostScript code for illustration..

```
showpage
%%EOF
%%EndDocument

EndEPSF                    % Restore state, cleanup
stacks
pgsave restore
showpage
%%EOF
```

Appendix: Changes Since Earlier Versions

This content of this document is exactly the same as the specification in Appendix H of the *PostScript Language Reference Manual, Second Edition*.

Changes Since Version 2.0

Detailed DSC comment descriptions have been left out of this specification. When developing an application that will support EPS files, the DSC version 3.0 (see the *PostScript Document Structuring Conventions Specifications* available from the Adobe Systems Developers' Association) should be used with this specification.

The following conditionally required DSC comments were added to this specification as of version 3.0:

%%Extensions:
%%LanguageLevel:
%%DocumentNeededResources:
%%IncludeResource:
%%Begin(End)Document:

Changes Relevant to Applications Producing EPS Files

To help avoid ambiguities, section 2, "Guidelines for Creating EPS Files," has been added. This new section has several guidelines for producing EPS files. Following these guidelines will help ensure that an EPS file can be reliably included in documents without causing any annoying side effects. Also, these new rules allow applications to easily determine if an EPS file is compatible with version 3.0 of the EPS file format. The following is an overview of the new guidelines:

- %%Begin(End)Preview: comments must bracket an EPSI preview.
- There is a list of illegal operators that must not be used in an EPS file.

- There is a list of restricted operators. If these operators are used in an EPS file, they must be used in accordance with the guidelines presented in Appendix I of the *PostScript Language Reference Manual, Second Edition*.

- The operand and dictionary stacks must be returned to the state that they were in before the EPS file was executed.

- It is strongly recommended that an EPS file make its definitions in its own dictionary or dictionaries.

- An EPS file must not rely on procedures defined outside of the server loop, such as procedures defined in the LaserPrep file.

Changes Relevant to Applications Importing EPS Files

To help clarify the responsibilities of an application including an EPS file, section 3, "Guidelines for Importing EPS Files," specifies the following new rules:

- The including application must define **showpage** as null.

- The application must prepare the graphics state for the EPS file.

- The application must give the EPS file a clear operand stack.

- The application must surround the included EPS file by the %%Begin(End)Document: comments.

Index

POSTSCRIPT™
Software From Adobe

Adobe Illustrator File Format Specification

Adobe Developer Support

Version 3.0

Draft
28 October 1992

Adobe Systems Incorporated

Corporate Headquarters
1585 Charleston Road PO Box 7900
Mountain View, CA 94039-7900
(415) 961-4400 Main Number
(415) 961-4111 Developer Support
Fax: (415) 961-3769

Adobe Systems Eastern Region
24 New England
Executive Park
Burlington, MA 01803
(617) 273-2120
Fax: (617) 273-2336

Adobe Systems Europe B.V.
Europlaza
Hoogoorddreef 54a
1101 BE Amsterdam Z-O, Netherlands
+31-20-6511 200
Fax: +31-20-6511 300

Adobe Systems Japan
Swiss Bank House 7F
4-1-8 Toranomon, Minato-ku
Tokyo 105, Japan
81-3-3437-8950
Fax: 81-3-3437-8968

PN LPS5007

Contents

Adobe Illustrator File Format Specification

1 Why This is a Draft Specification

This specification is still in draft form. We recognize that this specification is not yet as definitive as it should be, nor is it a clear, concise explantation of the Adobe Illustrator file format. However, we do feel that this document is suitable for those willing to use it as a reference while simultaneously examining real Adobe Illustrator files. In cases where the documentation and the sample file are in disagreement, the actual file should be used as the "correct" interpretation.

2 Introduction

This Technical Note describes the format ofAdobe Illustrator 3.x documents (files). An Adobe Illustrator document is a PostScript® language page description that conforms to the *Adobe Document Structuring Conventions Specification*, version 3.0. Any PostScript language interpreter can execute the page description to render the illustration on a display or the printed page. Page composition systems and other illustration-editing applications can also import such a document. Programs such as spoolers may parse page descriptions to extract information about resources that the document requires—fonts or procedure sets, for example.

Adobe Illustrator can store a document in *encapsulated PostScript file* (EPS) format; however, its default format is not an official EPS format. Other formats, explained in section section 14," include the EPS file format. An EPS file can include an image of the illustration suitable for previewing on specific computer platforms, and a user can select whether the preview image is in color or black and white. A page composition system can use a preview image for picture placement, scaling, cropping, and previewing.

Although an intimate knowledge of the PostScript language is not essential for understanding this document, you should be familiar with all terms used in the *Adobe Illustrator User Guide*, and several additional concepts:

- Adobe Illustrator descriptions are based on the path, as described in *PostScript Language Reference Manual, Second Edition*. Adobe Illustrator paths, however, are not as general as paths created for execution by the PostScript language.

- You can put opaque ink on the page either by *stroking* the path to trace it with a line of a given thickness (as with a pen), or by *filling* the path to put ink everywhere inside the path.

- The thickness of the line used to stroke the path, the color of the ink used, and so forth, are attributes of the PostScript *graphics state*. A PostScript language program can change the graphics state and thus control how a path is marked. In Adobe Illustrator, fill and stroke attributes maintain their own states. Whenever the graphic state attributes that would affect filling or stroking are changed, Illustrator appropriately modifies procedures that carry out the fill or stroke.

- Text is a special case of the more general drawing facilities of the PostScript language. Pre-defined PostScript programs (called *fonts*) render individual characters by constructing and then filling or stroking a path.

- Coordinate values written out for objects in the Adobe Illustrator file are offsets from the ruler origins. Generally, the origin is in the lower left-hand corner of the space, the *x*-axis extends horizontally to the right, and the *y*-axis extends vertically upward. However, changes in the ruler origin can modify this. The length of a unit along each axis is $^1/_{72}$ inch, which approximates a printer’s point ($^1/_{72.27}$ inch).

For additional information about the PostScript language, see the *PostScript Language Reference Manual*, *Second Edition*, Addison-Wesley, ISBN 0-201-18127-4. A full discussion of Document Structuring Conventions and Encapsulated PostScript Files also appear as appendices in that document.

3 Overview

Adobe Illustrator creates documents (files) that conform to Adobe Systems’ *document structuring conventions*. A PostScript language document description that minimally conforms to the Document Structuring Conventions has two main parts: a *prolog* and a *script*. The prolog portion encapsulates information needed by other programs to interpret the file, such as the bounding box that contains all marks on the page. It also contains lists of resources, such as fonts, that the pages may need, and PostScript language procedure definitions and the variables that are used in the document’s page descriptions. Documents that conform to the *Document Structuring Convention*s do not execute any PostScript code in the prolog (other than definitions) and should not define new procedures or global variables in the script (although they may well modify the behavior of procedures defined in the prolog or set the value of global variables).

Adobe Illustrator uses a subset of the full PostScript language to describe the graphic elements on a page. The subset is purely declarative; properly setting the position or attributes of graphic elements on the page does not require complex computation in the PostScript language. Pages expressed this way print or image faster, and generally are smaller in size. The language subset is defined in *Resources* (sometimes called *Procedure Sets* or *proc sets*), logically grouped together in the prolog with explicit methods for their initialization and termination.

Note *Don't confuse the term "resources" as used in Adobe Illustrator files with a file's resource fork as used in the Macintosh environment, resources as defined in PostScript Language Reference Manual, Second Edition, or procedure sets encapsulated as modules of PostScript language code.*

After the prolog, the main body of the document is the *script*. A script has three logical sections: a *Setup* sequence that initializes and activates the Resources defined in the prolog, a sequence of *descriptive operators*, and a *Trailer* that deactivates Resources. The script holds the operators, which are sequences of graphic elements and which are written in the language subset as defined in the prolog. These sequences consist of collections of data elements, graphic attribute definitions, and calls to the procedures defined in the prolog's Resources.

The prolog part of a document contains information needed by other programs that may interpret the file. Illustrator includes the different parts of the prolog as needed by the file. For example, the prolog contains the PostScript language definition of procedures and variables used in the individual page descriptions. It can contain a set of comments collectively referred to as a "header" that can provide certain information to Illustrator and other programs that read and interpret those comments. The prolog also defines the collection of PostScript procedures that implement the operators of the special language subset. Adobe Illustrator 3.x places no prolog in its default "no header" save format; it places a header in its other save formats. The main body of the file is the script which describes the document by using the illustration language.

Generally speaking, Illustrator works in the following way: graphic state attributes are set up, rendering routines prepared, the path is expressed, and then the generic procedures are called the carry out the fill, stroke, or other rendering.

3.1 Comments

Stylized comments form the document structuring information in a PostScript program.The PostScript langugage treats as a comment any line where the first non-whitespace character is **%**. Apart from the first comment in a file, the structuring comments all have the form

```
%%Keyword{: arguments}
   or
   %AI3_Keyword{: arguments}
```

Many structuring comments require information in addition to the keyword. This information is separated from the keyword by a colon and continues on to the newline character that terminates the comment.

3.2 Syntax

The Adobe Illustrator document format can be described using BNF (Backus-Naur form) notation:

```
<xyz>::=                abc <def> ghi |
                        <k> j
```

A token enclosed in angle brackets names a class of document component, while plain text appears verbatim or with some obvious substitution. The grammar rules have two parts. On the left of the ::= definition symbol is the name of a class of component. In the example above, the class is *xyz*. On the right of the definition symbol is a set of one or more alternative forms that an *xyz* component might take in the document. The alternative forms are separated by the vertical bar character (|).

Each line on the right-hand side of the definition corresponds to one line in the document. If the only content on the right-hand side is another class name, then the line may represent more than one line in the document. Single letter components, such as <*A*>, refer to the corresponding illustration language operator *A*. The notation *{…}* means that the items enclosed in braces are optional. If an asterisk follows the braces, the objects inside the braces may be repeated *zero* or more times. The notation $<\ldots>^+$ means that the items enclosed within the brackets may be repeated *one* or more times.

For example, using BNF notation, you can described the overall structure of an Adobe Illustrator document as

```
<document> ::=        <prolog>
                      <script>

<prolog> ::=          %!PS-Adobe-3.0 EPSF-3.0
                      <header comments>
                      %%EndComments
                      {<proc set>}*
                      %%EndProlog

<script> ::=          <setup>
                      {<object>}*
                      {<page trailer>}
                      <document trailer>
                      %%EOF
```

The full document syntax structure appears in section section 17.” The syntax and semantics of the individual operators of the illustration language are defined in later sections of this document. Each operator definition follows this form:

$\text{operand}_1 \ldots \text{operand}_m$ **op –**

The functionality of the **op** operator follows here.

This notation means that the operator **op** takes operands from *1* through *m* and performs some operation. Each operand is characterized either by its data type (for example, *integer*) or a more meaningful name, such as *linewidth*. In the latter case, the range of legitimate values appears in the description. A dash (–) on the left of the operator indicates that an operator requires no operands; a dash on the right indicates that the operator leaves nothing on the stack.

3.3 Differences Between Versions

There are several versions of Adobe Illustrator available. These include *Adobe Illustrator 3.x, Adobe Illustrator for Windows Version 4.x,* and *Adobe Illustrator Japanese Edition*. This document describes the file format for Adobe Illustrator 3.x; for most purposes, the format for Adobe Illustrator for Windows Version 4.x is the same as that for 3.x, with differences noted in section section 15.” Adobe Illustrator 88 and Adobe Illustrator Japanese Edition are described in the Adobe Technical Note **XXX-XXX**.

Where differences bear on which operands to use for particular operators or which comments to include in a header, those differences are indicated in the descriptions of the individual components of the document. Parts of the document or individual operators that are used exclusively by one or another versions are so marked. Generally, Adobe Illustrator software ignores comments and operators that are meant for other versions.

This document uses the following version abbreviations:

Adobe Illustrator, version 3.x	AI3
Adobe Illustrator for Windows Version 4.x	AIWin
Adobe Illustrator NeXT, version 3.x	AINeXT
Adobe Illustrator Japanese Edition	AIJE
Adobe Illustrator 88	AI88

4 Prolog

The syntax for an Adobe Illustrator 3.x document prolog is:

```
<prolog> ::=        %!PS-Adobe-3.0 EPSF-3.0
                    <header comments>
                    %%EndComments
                    {<proc set>}*
                    %%EndProlog
```

%!PS-Adobe-3.0 EPSF-3.0

The first line of the file is a unique comment that identifies the version of the Document Structuring Conventions to which the document conforms. In this case, the file conforms to version 3.0. The first line may also specify that the file conforms to a version of the Encapsulated PostScript File format (EPSF). Both numbers must correspond to the specific versions used in writing out the file.

%%BeginProlog

The **%%BeginProlog** comment marks the beginning of the prolog section.

%%EndProlog

The **%%EndProlog** comment marks the end of the prolog section.

4.1 Header

The header for the prolog body of an Adobe Illustrator document follows the version-identifying first line of the file. The syntax for the prolog header is

```
<header> ::=           <header comments>
                       %%EndComments
```

%%EndComments

The **%%EndComments** comment marks the end of the header part of the prolog.

The sequence of header comments is a subset of those listed in Figure 1. The syntax for each comment is described informally. Almost all header comments are optional. Comments required by Adobe Illustrator in all documents are marked **[Required]**. Some comments are required only if a specific feature is used in an illustration. Such comments are marked **[As Necessary]**. Those comments that are specific to Adobe Illustrator version 3.x have the form

```
%AI3_Keyword{: arguments}
```

Figure 1 . *List of typical Adobe Illustrator header comments; comments can vary from file to file*

```
%%Creator: Adobe Illustrator(TM) 3.0.1
  %%For: (username) (organization)
  %%Title: (illustration title)
  %%CreationDate: (date) (time)
  %%BoundingBox: llx lly urx ury
  %%DocumentProcessColors:  keyword
  %%DocumentFonts:  font…
  %%+font…
  %%DocumentFiles:  filename
  %%+filename…
  %%DocumentSuppliedResources:  procset Adobe_packedarray ver-
  sion revision
  %%+ procset Adobe_cmykcolor version revision
  %%+ procset Adobe_cshow version revision
  %%+ procset Adobe_customcolor version revision
  %%+ procset Adobe_pattern_AI3 version revision
  %%+ procset Adobe_typography_AI3 version revision
  %%+ procset Adobe_IllustratorA_AI3 version revision
  %AI3_ColorUsage:  keyword
  %AI3_TemplateBox: llx lly urx ury
  %AI3_TemplateFile:  pathname
  %AI3_TileBox:  llx lly urx ury
  %AI3_DocumentPreview:  previewtype
```

The individual lines in the header are specified as follows.

%%Creator: Adobe Illustrator(TM) *version*

The **%%Creator** comment identifies the application that generated the PostScript language document. The *version* number is arbitrary text terminated by a newline character.

%%For: (*username*) (*organization*)

The **%%For** comment identifies the user who created the file and the organization to which the user belongs. Both *username* and *organization* are valid PostScript language strings. The PostScript language string escape sequences for including characters outside the printable ASCII character set and for representing characters such as *(* and *)* and other special characters are discussed in *PostScript Language Reference Manual, Second Edition*.

%%Title: (*illustration title*)

The **%%Title** comment provides an arbitrary text title for the document. The title is a valid PostScript language string.

%%CreationDate: (*date*) (*time*)

The **%%CreationDate** comment gives the date and time that the document was created. The variables *date* and *time* are valid PostScript language strings.

%%BoundingBox: *llx lly urx ury*

[Required]. The **%%BoundingBox** comment specifies the imaginary box that encloses all marks painted on the page. Specify the integer coordinates in the default user coordinate system. Negative numbers are allowed.

%%DocumentProcessColors: *keyword*

The **%%DocumentProcessColors** comment specifies which of the process colors identified by the keywords **Cyan**, **Magenta**, **Yellow**, and **Black** the document uses. This comment is used primarily by programs producing color separations.

%%DocumentCustomColors: (*customcolorname*)
%%+ (*customcolorname*)

[As Necessary]. The **%%DocumentCustomColors** comment enumerates the names of the custom colors used in the document. The names are valid PostScript language strings enclosed in parentheses. For example, the PANTONE® colors are identified by names such as **(PANTONE 156 CV)**. You may continue the list of custom color names on subsequent lines, each of which must begin with the **%%+** prefix.

%%CMYKCustomColor: *cyan magenta yellow black (customcolorname)*
%%+ *cyan magenta yellow black (customcolorname)*

[As Necessary]. The **%%CMYKCustomColor** comment specifies an approximation for the named custom color in terms of the four components of process color: *cyan*, *magenta*, *yellow*, and *black*. Each component value must be a real number in the range 0.0 to 1.0. The component values are analogous to the arguments to the PostScript language operator **setcmykcolor**.

%%DocumentFonts: *font…*
%%+*font…*

[As Necessary]. The **%%DocumentFonts** comment enumerates the names of the PostScript language font programs that the document uses. Fonts listed in the **%%DocumentFonts** comment are also included in any files which are themselves included (placed) within an Adobe Illustrator document. Omit this comment if the document uses no fonts.

You may need to download a font to the PostScript device before it can properly execute a document description.

%%DocumentFiles: *filename*
%%+*filename…*

[As Necessary]. The **%%DocumentFiles** comment names the files that a program must import to render the illustration. Another comment (**%%IncludeFile**) marks the site within the illustration at which the file is needed. Omit this comment if no files are to be imported into the document. See section 9” for more information on including files.

```
%%DocumentSuppliedResources:  procset Adobe_packedarray version revision
                              %%+ procset Adobe_cmykcolor version revision
                              %%+ procset Adobe_cshow version revision
                              %%+ procset Adobe_customcolor name version revision
                              %%+ procset Adobe_pattern_AI3 version revision
                              %%+ procset Adobe_typography_AI3 version revision
                              %%+ procset Adobe_IllustratorA_AI3 version revision
```

[AI3]. These comments (the above list is only an example) indicate that the named procedure sets and resources are both required *and* defined by the document. The **%%+ procset** construction indicates a continuation of the **%%DocumentSuppliedResources** comment.

Comments appear only for the actual procedure sets needed by the illustration; they are not present when the file is saved in its “no-header” form.

%%DocumentNeededResources

[AI3] This comment tells what procsets are required by the document that are not included within it.

%%IncludeResource

[AI3] This comment lists a specific resource to include in a document. It is present only when the file is saved in its “no-header” format.

%AI3_ColorUsage: *keyword*

The **%AI3_ColorUsage** comment indicates whether the document uses only black or colored ink, indicated by the keyword **Black&White** or **Color**, respectively.

[AIWin, AI88, AIJE]. This comment is named **%%ColorUsage**.

%AI3_TemplateBox: *llx lly urx ury*

The **%AI3_TemplateBox** comment specifies the bounding box that encloses all samples in the document’s template. For more information on templates, see *Adobe Illustrator User Guide.* Specify the coordinates as integers or reals in the default user coordinate system. Each sample in the template is assumed to be $^1/_{72}$ inch square. The width *(urx–llx)* and height *(ury–lly)* of the template box must be integers. If a document has no template, the width and the height of the template box must be zero.

When Adobe Illustrator opens a document, it centers the coordinate ((*llx* + *urx*)⁄2, (*lly* + *ury*)⁄2) in the drawing area.

[AIWin, AI88, AIJE]. This comment is named **%%TemplateBox**.

%AI3_TemplateFile: *pathname*

[As necessary]. The **%AI3_TemplateFile** comment specifies the template for the illustration. If no name is given, no template is used. The format for specifying the template is:

<volume name>::<directory id (a number)>:<filename>

%AI3_TileBox: *llx lly urx ury*

The **%AI3_TileBox** comment is used only on the Macintosh version of Adobe Illustrator. It specifies the bounding box of the imageable area of the current page size. See *Adobe Illustrator User Guide* for more information about tiles and the drawing area. The initial ruler position is centered in this box.

[AIWin, AI88, AIJE]. This comment is named **%%TileBox**.

4.2 Artwork and Ruler Origin

All artwork elements, as well as the Bounding Box, Template Box, and Tile Box, are written out in coordinates relative to the *ruler origin*, with *y* increasing up and *x* increasing to the right, and bounds in the order left, bottom, right, top.

The template, if there is one, is always centered on the artboard. If there is no template associated with the artwork, the **%AI3_TemplateBox** comment describes a degenerate box positioned at the center of the artboard. Since it is written out in ruler-relative coordinates, the center of the template bounding box can be used to establish the ruler origin by measuring backwards from the center of the current artboard (that is by measuring *x* to the left of the center of the template bounding box and *y* down from the center). It is done this way since the size of the artboard may change between the version of Illustrator a file is saved with and the version it is read back in under. In such a situation, it is the *centers* of the two artboards that need to be aligned.

That is, when the file is opened, the Template Box rectangle is read in, and then the ruler origin is calculated as:

$$x = (\text{artboard width} - \text{templateBox.left} - \text{templateBox.right})/2$$

$$y = (\text{artboard height} + \text{templateBox.top} + \text{templateBox.bottom})/2$$

(This *x*,*y* is in Macintosh space, where *y* increases down, unlike the Illustrator file format, where y increases up.)

The position of the ruler, of course, is only really meaningful inside Illustrator or another illustration editing program that wishes to import Illustrator files and keep the ruler position intact. For applications that do not care about Illustrator’s ruler position, it is sufficient to choose as an origin any point pertinent to the importing application, such as one of the corners of the bounding box, and apply to all the points in the artwork the translation that would take that point to 0,0.

5 Script Setup

The syntax for the script setup section of an Adobe Illustrator 3.x document is

```
<script> ::=             <setup>
                         {<object>}*
                         {<page trailer>}
                         <document trailer>
                         %%EOF

<setup> ::=              %%BeginSetup
                         {%%IncludeFont: font}*
                         {<proc set init>}*
                         <font encoding>
                         <pattern defs>
                         %%EndSetup

<page trailer> ::=       %%PageTrailer
                         gsave annotatepage grestore showpage

<document trailer> ::=   %%Trailer
                         {<proc set termination>}*
```

The following sections describe the individual components of the setup section of an Illustrator document.

5.1 Specifying Particular Fonts

The comment **%%IncludeFont** specifies a font that appears in he document. Adobe Illustrator 3.x checks to see whether that font is available and uses it if it is. If the font is not available, Adobe Illustrator uses another font.

5.2 Initializing Resources

Adobe Illustrator customarily *initializes* those resources (proc sets) required by the document. A corresponding *termination* appears in the document trailer.

5.3 Fonts and Encodings

The mapping between ASCII characters and glyphs in a font is different from the standard mapping used in a PostScript font. Therefore, to print a document correctly, the mapping must be changed for each PostScript font used in an Adobe Illustrator document. The action of altering the mapping between character codes and glyphs is called *re-encoding* the font.

The syntax for re-encoding a font in an Adobe Illustrator document is

```
<font encoding> ::=     [
                        {<re-encoding pairs>}*
                        <Te>
                        {<re-encoding>}*

<re-encoding> ::=       %AI3_BeginEncoding
                        newFontName oldFontName
                        <TZ>
                        %AI3_EndEncoding <font type>

<font type> ::=         AdobeType|TrueType
```

The **TE** operator sets the standard encoding for the platform on which the Illustrator file is being executed. The **TZ** operator performs the re-encoding. Once encoding has been specified, the **Tf** operator can specify the font name and the font size.

[<encoding pairs> **TE**

The **TE** operator sets the standard platform font encoding. Note that there is no right bracket following the parameter.

[newFontname oldFontName direction fontScript useDefault **TZ**
[encodingPairs newFontName oldFontName direction fontScript useDefault **TZ**
[newFontName oldFontName direction fontScript useDefault [w0 w1…wn] **TZ**
[encodingPairs newFontName oldFontName direction fontScript useDefault [w0 w1…wn] **TZ**

The **TZ** operator creates a new font from an existing font by changing portions of the new font's encoding vector. The first two forms are for Type 1 font programs; the second two forms are for Multiple Master typefaces. The forms with the *encodingPairs* operand are used when changing font encoding. These encodings are platform-specific to the Macintosh computer; the NeXT computer, for example, may not regularly need encodings because it uses the Display PostScript System.

The *encodingPairs* operand is a list of encoding numbers and literal glyph names organized as follows:

$\texttt{code}_1$ $\texttt{/name}_{11}$ $\texttt{/name}_{12}$ … $\texttt{/name}_{1j}$
  $\texttt{code}_2$ $\texttt{/name}_{21}$ $\texttt{/name}_{22}$ … $\texttt{/name}_{2k}$
  …
  $\texttt{code}_n$ $\texttt{/name}_{n1}$ $\texttt{/name}_{n2}$ … $\texttt{/name}_{nl}$

where each *code* is in the range 0 to 255 and each *name* is the literal glyph name. The **/** preceding each name is the syntax used to distinguish a PostScript language literal name from an executable name. This list describes a set of sequences of glyph names to install in the new encoding vector. Each sequence begins with the character index of the first name to be replaced. Subsequent names are replaced up to the next character index entry in *encodingPairs*, at which point a new sequence of replacement names begins, starting with the one at the new character index.

The *newFontName* and *oldFontName* operands are the PostScript names for the new font and the original font. These names must be the same as the names given in the **%%BeginEncoding** comment. The *direction* operand is zero.

Note *For versions of Adobe Illustrator other than Adobe Illustrator 3.x and Adobe Illustrator Japanese Edition, there is no* direction *operand.*

For composite fonts (such as Japanses language fonts) the *direction* operand is 0 for horizontal writing and 1 for vertical writing. The *fontScript* operand is 0 for Roman typefaces and 1 for composite typefaces. For composite fonts the [*encodingPairs* list must have a single left bracket.

The *defaultEncoding* operand controls whether the **TE** encoding is used (1) or not (0). If the font is a Multiple Master typeface, the final array operand is the weightVector of the Multiple Master instance.

Figure 2 shows how to use the **TZ** operator. The example derives a new font named *_Times-Roman* from the original *Times-Roman* font. It replaces three sequences of characters within the encoding vector; the first one-character sequence is number 39, the second one-character sequence is number 96, and the third sequence replaces the characters numbered 128 and above.

Figure 2 . *Re-encoding Times Roman with the TZ operator*

```
%%BeginEncoding: _Times-Roman Times-Roman
   [
   /_Times-Roman/Times-Roman 0 0 1 TZ
   %%EndEncoding
```

5.4 Pattern Definition

The script setup section of a document defines the patterns used by Adobe Illustrator. A pattern is essentially just another Adobe Illustrator illustration that can be drawn repeatedly. You cannot use placed files nor graph objects within a pattern, but patterns can include paths and text. Therefore, parts of the description of how patterns are defined necessarily refers to the description of how an illustration is described in the document script section.

Each pattern is defined by a rectangle used to *tile* the drawing area. The illustration within that rectangle constitutes the pattern used when a path is stroked or filled with a pattern.

The syntax for a pattern is

```
<pattern defs> ::=      {<pattern>}*

<pattern> ::=           %AI3_BeginPattern:  (patternname)
                        <E>
                        %AI3_EndPattern
```

(patternname) llx lly urx ury [<layer list>] **E**

The **E** operator defines a new pattern called *patternname* using the *layer list* for which the bounding box is specified by (*llx, lly*) and (*urx, ury*).

Section section 6” describes how a general illustration is composed of layers, each of which is drawn on top of lower layers which appear earlier in the sequence. Because a pattern is really just a mini-illustration, it too is composed of layers. The syntax for the list of layers is

```
<layer list> : :=       {<layers>}*

<layer> ::=             <@>
                        <&>
```

Each layer of the pattern consists of two parts. The first part defines the color to be used for filling and stroking the pattern. The second part defines the other style parameters and the paths for drawing the pattern.

(colordefinition) **@**

The **@** operator defines the color and overprinting style for the associated layer in the pattern. The *colordefinition* parameter begins with a specification of the overprinting option. For more information on overprinting, see the definitions of operators **O** and **R** in section section 6.3.” The filling or stroking color is then defined using the simple gray operators (**g** and **G**), the process color operators (**k** and **K**), or the custom color operators (**x** and **X**). All color operators are defined in section section 6.3.”

(tiledefinition) **&**

The **&** operator defines the tile for the pattern layer that includes the drawing styles and illustration components. This is identical to the representation of objects in the document script except that the color components and both parts of the object are specified separately as PostScript language strings, which are enclosed in parentheses. The use of strings limits the size of a pattern layer definition to 64K bytes.

Whenever a pattern background is filled or stroked, the first layer of the pattern defines the background for the tile. If the pattern tile rectangle is filled, then you must first use the special _ (underbar) operator to specify a fill. If the pattern tile rectangle is stroked, then normal path construction of the rectangle specifies the pattern tile to stroke. Breaking down the filling and stroking of the pattern tile results in performance optimization when imaging.

_ _

The _ (underbar) operator signals the pattern machinery that the tile rectangle for the path is to be filled with the fill color previously specified to the @ operator. Figure 3 shows an example pattern definition.

Figure 3 . *An example pattern definition*

```
%%BeginPattern:  (no vegetation)
   (no vegetation) 105 561.875 138 594.875 [
   (0 O 0 R 0.03 0,05 0.15 0 (PANTONE 468 CV) 0 x 0.03 0.05 0.15 0
   (PANTONE 468 CV) 0 X) @
   _ &
   (0 O 0 R 0.125 0.25 0.525 0 (PANTONE 465 CV) 0 x 0.125 0.25
   0.525 0 (PANTONE 465 CV) 0 X) @
   (
   0 i 0 J 0 j 1 w 4 M []0 d
   %%Note:
   105 561.875 m
   105 594.875 L
   138 594.875 L
   138 561.875 L
   105 561.875 L
   s
   ) &
   (0 O 0 R 0 0.15 0.4 0 (PANTONE 156 CV) 0 x 0 0.15 0.4 0 (PANTONE
   156 CV) 0 X) @
   (
   0 i 0 J 0 j 0.3 w 4 M []0 d
   %%Note:
   105 599 m
   105 560 I
   S
   …
   138 599 m
   138 560 I
   S
   ) &
   ] E
   %%EndPattern
```

The example in Figure 3 defines a pattern called “no vegetation” where the pattern tile is both filled and stroked.

First is the pattern name, the bounding box for the pattern tile, then the layer list. The first item in the layer list specifies the custom color *PANTONE 468 CV* as the fill color of the pattern tile. The _ operator specifies a fill of the pattern tile. The next layer in the pattern specifies a stroke of the pattern tile. The custom color *PANTONE 465 CV* is the stroke color. Following the color specification is the drawing of the pattern tile itself. See section section 6.2” for a description of the style options selected at the beginning of the tile definition. Each path in the tile layer is then specified with a sequence of **m** (**moveto**) and **l** (**lineto**) operations. Each path in the layer is stroked by the **S** operator.

Once the pattern tile is drawn and stroked, the elements of the pattern follow; in this example, they are vertical lines which are all stroked with the same color.

6 Script Body

An illustration is composed of a sequence of graphic elements. The PostScript imaging model is based on opaque ink so that elements later in the sequence are effectively “on top of” other elements earlier in the sequence. Thus, later elements can obscure earlier elements.

Fill and stroke attributes are *state-based*; that is, once set, they remain set until changed.

The syntax for the sequence of elements is

```
<object> ::=              {<A>}(object locking)
                          <path object>|
                          <path mask>|
                          <composite object>|
                          <text object>|
                          <placed art object>|
                          <subscriber object>|
                          <graph object>|
                          <PostScript document>

<path object> ::=         <paint style>
                          <path geometry>
                          <path render>|<*> (<*> indicates
                          guide operator)

<path mask> ::=           <paint style>
                          <path geometry>
                          <h>|<H>
                          <W>
                          <path render>

<composite object> ::=
                          <group object>|
                          <group with a mask>|
                          <compound path>|
                          <compound path mask>|
                          <wraparound group>

<group object> ::=        <u>
                          <object>+
                          <U>
```

```
<group with a mask> ::=
                        <q>
                        {<object>}*
                        {<masked object>}*
                        <Q>

<masked object> ::=     <mask>|<object>

<mask> ::=              <path mask>|<compound path mask>

<compound path> ::=     <*u>
                        <compound path element>+
                        <*U>

<compound path element> ::=
                        <path object>|<compound group>

<compound group> ::=
                        <u>
                        <compound path element>+
                        <U>

<compound path mask> ::=
                        <*u>
                        <compound path mask element>+
                        <*U>

<compound path mask element> ::=
                        <path mask>|<compound mask group>

<compound mask group> ::=
                        <compound mask bottom group>|
                        <compound mask non-bottom group>

<compound mask bottom group> ::=
                        {<A>}
                        <q>
                        <path mask>+
                        <Q>

<compound mask non-bottom group> ::=
                        {<A>}
                        <u>
                        <compound mask group>+
                        <U>
```

The following sections explain the individual operators for describing graphic objects.

6.1 Locked Object Operator

flag **A**

The **A** operator specifies whether the object defined immediately after the operator can be selected when editing the document with Adobe Illustrator. The *flag* operand may be either 0 or 1. If *flag* is 0, the object may be selected for editing. If *flag* is 1, the object is “locked” and may not be selected. This state remains in force for every subsequent element unless specifically changed.

6.2 Graphics State Operators

[array] phase **d**

The **d** operator is equivalent to the PostScript language **setdash** operator. It sets the dash pattern parameter in the graphics state, to control the dash pattern of subsequently stroked paths. The *array* of values specifies distances in user space for the length of dashes and gaps, respectively, in the dash pattern. The *phase* operand determines the phase of the dash pattern with respect to the start of the path. It is specified as a distance in user space into the pattern at which to begin marking the path. The initial dash pattern is a solid line.

Note *Adobe Illustrator does not provide an interface for users to adjust this phase parameter. Adobe Illustrator preserves this phase for documents that the user edits and saves.*

flatness **i**

The **i** operator is equivalent to the PostScript language **setflat** operator, which sets the flatness parameter in the graphics state. The *flatness* parameter specifies the accuracy or smoothness with which curves are rendered as a sequence of flat line segments. Specifically, it sets the maximum permitted distance in device pixels between the mathematical path and a given straight line segment. The default value for the *flatness* parameter is 0.0. If *flatness* is specified as 0, *flatness* is set by Adobe Illustrator to the *flatness* parameter in effect when the prolog was executed; in most cases, that is the device’s default flatness. This may be the device’s default flatness, it may be a value you have entered, or it may be a value inherited from an encapsulating context. Acceptable range is 0 to 100.

flag **D**

The **D** operator determines the *winding order* of the object. The PostScript language fills areas to the left of the path direction. The operand *flag* is 0 for a clockwise path direction and 1 for a counter-clockwise direction.

linejoin **j**

The **j** operator is equivalent to the PostScript language **setlinejoin** operator, which sets the line join parameter in the graphics state. This parameter specifies the shape to put at corners in paths when they are stroked. The *linejoin* parameter may be 0 for mitered joins, 1 for round joins, and 2 for beveled joins. The initial *linejoin* is 0.

linecap **J**

The **J** operator is equivalent to the PostScript language **setlinecap** operator, which sets the line cap parameter in the graphics state. If *linecap* is 0, Illustrator uses butt end caps and squares off line ends. If *linecap* is 1, it uses round end caps. If *linecap* is 2, it uses projecting square end caps. The projection extends beyond the end of the line by a distance which is half the line width. The initial *linecap* value is 0.

miterlimit **M**

The **M** operator is equivalent to the PostScript language **setmiterlimit** operator, which sets the miter limit parameter in the graphics state. The *miterlimit* operand must be a real number greater than one. When you have specified mitered joins and two line segments meet at a sharp angle, it is possible for the miter to extend far beyond the thickness of the line stroking the path. The miter limit imposes a limit on the ratio of the length of the miter to the line width. When the limit is exceeded, the file prints with a bevel join instead of a miter. The initial miter limit is 4.

linewidth **w**

The **w** operator is equivalent to the PostScript language **setlinewidth** operator, which sets the line width parameter in the graphics state. This parameter controls the thickness of the line used to stroke a path and is specified as a distance in user space. The initial *linewidth* is 1.0. A line width of 0 is acceptable; when Illustrator prints the file, this is interpreted as the thinnest line width that can be rendered at device resolution (not recommended on high-resolution devices because the line may be nearly invisible).

6.3 Color Operators

The settings for color operators and gray scale can extend to four decimal places.

gray **g**

The **g** operator specifies the gray tint to use for filling paths. The *gray* operand must be a real number between 0.0 (black) and 1.0 (white).

gray **G**

The **G** operator is similar to the **g** operator, but specifies the gray tint to use for stroking paths. The *gray* operand must be a real number between 0.0 (black) and 1.0 (white).

cyan magenta yellow black **k**

The **k** operator is equivalent to the PostScript language **setcmykcolor** operator. It specifies the color to use for *filling* paths. Each operand must be a real number between 0.0 (minimum intensity) and 1.0 (maximum intensity). If the **setcmykcolor** operator is not defined by the PostScript interpreter (except in the case of creating separations), the Adobe Illustrator prolog defines it in terms of the original **setrgbcolor** operator by transforming the operands as follows.

```
red = 1 – min(1, cyan + black)
   green = 1 – min(1, magenta + black)
   blue = 1 – min(1, yellow + black)
```

The PostScript interpreter automatically performs the conversion from red, green, and blue to gray for a monochrome output device using the following formula.

```
gray = 0.3 × red + 0.59 × green + 0.11 × blue
```

cyan magenta yellow black **K**

The **K** operator is similar to the **k** operator, but specifies the color to use for stroking paths.

cyan magenta yellow black (name) gray **x**

The **x** operator defines a custom color for filling paths. The *cyan*, *magenta*, *yellow*, and *black* operands are interpreted in the same way as for the **k** and **K** operators. Adobe Illustrator treats the *gray* operand the same as for the **g/G** operators, and specifies the screen fraction of the custom color in the range 0.0 to 1.0. The *name* operand is a valid PostScript language string that names the custom color. For example:

```
0.45 0 0.25 0 (PANTONE 570 CV) 0 x
```

The first four operands are CMYK values. name is the name of the color; this is popped off in execution. The last operand (*gray*) is the tint value.

A user can specify the tint value in percentages from 0 to 100%. The value is scaled to the range of 0 to 1 and then subtracted from 1 to determine what is to be written out in the PostScript language call. A custom color's CMYK values are each multiplied by the tint value.

cyan magenta yellow black (name) gray X

The **X** operator is similar to the **x** operator, but specifies the custom color to use for stroking paths.

(patternname) p_x p_y s_x s_y angle rf r k ka [a b c d t_x t_y] **p**

The **p** operator specifies the pattern for subsequent fill operations. The *patternname* operand names the pattern as defined in the script setup sequence (see section section 5"). The remaining operands specify the transformations—in order—to be applied to the pattern before using it to fill a path.

p_x p_y Specify the offset from the ruler origin of the origin to be used for tiling the pattern. Each distance specified in points.

s_x s_y Specify the scale factors to be applied to the *x* and *y* dimensions, respectively, of the pattern.

angle Specifies the angle in counterclockwise degrees to rotate the pattern.

rf Flag indicating whether to apply a reflection to the pattern (1 = *true*, 0 = *false*).

r Specifies the angle of the line from the origin about which the pattern is reflected. Used if the *rf* operand is non-zero.

k Specifies the shear angle.

ka Specifies the shear axis.

[a b c d t_x t_y] Specifies the initial matrix to which all other pattern transformations are to be applied. This matrix describes transformations that are not otherwise expressible as the single combination of the other transformations. (See the description of the *Transform Pattern Style* sub-menu of the *Paint* menu in the *Adobe Illustrator 3.0 User Guide*.)

(patternname) p_x p_y s_x s_y *angle rf r k ka [a b c d* t_x t_y*]* **P**

The **P** operator is similar to the **p** operator, but specifies the pattern for use in stroking paths.

flag **O**

The **O** operator specifies whether to use overprinting when filling a path. If *flag* is 1 overprinting is used; otherwise *flag* must be 0. See *Adobe Illustrator 3.0 Color Guide* for a discussion of overprinting in Adobe Illustrator 3.x documents.

flag **R**

The **R** operator is similar to the **O** operator, but specifies whether to use overprinting when stroking a path.

6.4 Group Operators

Two operators support Adobe Illustrator’s ability to combine separate graphic elements into a single object.

– u

The **u** operator marks the beginning of a sequence of elements to be grouped into a composite object. All subsequent graphical elements in the script—including other groups, and up to a matching **U** operator—are included in the group.

– U

The **U** operator marks the end of a sequence of elements to be grouped into a composite object. A **u** operator must precede the **U** operator.

6.5 Paths

In the PostScript language, you draw by constructing a path and then filling or stroking it. This section defines the path operators. You can construct only one path (the *current path*) at a time. The current path is initially empty. The painting operators reset it to empty after execution.

6.6 Path Construction Operators

A path is constructed by appending segments which are either straight lines or Bézier curves. The last point on a segment is called the *current point*; new segments are always appended to the current point. The first operator on a path must be **m** to establish an initial current point.

As each new segment is appended to the path, Adobe Illustrator marks the new current point as either a smooth point or a corner point. If the point is smooth, Illustrator assumes collinearity of the point and the two associated Bézier direction points of the segments connected by the point. If the point is a corner point, there is no assumed constraint. You can think of a straight line segment as a “degenerate” Bézier curve in which the direction points are coincident with the end points.

The syntax for a path is as follows.

```
<path geometry) ::=    <m>
                       {<path operator>}*

<path operator> ::=    <l>|<L>|<c>|<C>|<v>|<V>|<y>|<Y>

<path render> ::=      <N>|(closepath; no fill no stroke)
                       <n>|(neither fill nor stroke)
                       <F>|(fill)
                       <f>|(closepath; fill)
                       <S>|(stroke)
                       <s>|(closepath; stroke)
                       <B>|(fill and stroke)
                       <b>(closepath; fill and stroke)
```

x y **m**

The **m** operator is equivalent to the PostScript language **moveto** operator. It changes the current point to *x, y*, omitting any connecting line segment. A path must have **m** as its first operator.

x y **l**

The **l** (lowercase L) operator appends a straight line segment from the current point to *x, y*. The new current point is a smooth point.

x y **L**

The **L** operator is similar to the **l** operator, but the new current point is a corner.

x1 y1 x2 y2 x3 y3 **c**

The **c** operator appends a Bézier curve to the path from the current point to x_3, y_3 using x_1, y_1 and x_2, y_2 as the Bézier direction points. The new current point is a smooth point.

x1 y1 x2 y2 x3 y3 **C**

The **C** operator is similar to the **c** operator, but the new current point is a corner.

x2 y2 x3 y3 **v**

The **v** operator adds a Bézier curve segment to the current path between the current point and the point x_3, y_3, using the current point and then x_2, y_2 as the Bézier direction points. The new current point is a smooth point.

x2 y2 x3 y3 **V**

The **V** operator is similar to the **v** operator, but the new current point is a corner.

x1 y1 x3 y3 **y**

The **y** operator appends a Bézier curve to the current path between the current point and the point x_3, y_3 using x_1, y_1 and x_3, y_3 as the Bézier direction points. The new current point is x_3, y_3 and is a smooth point.

x1 y1 x3 y3 **Y**

The **Y** operator is similar to the **y** operator, but the new current point is a corner.

6.7 Path Painting Operators

Each of the path painting operators consumes the *current path* and resets it to empty.

– **N**

The **N** operator neither fills nor strokes the current path, leaving it as an open path (see the **F/f** and **S/s** operators). Paths that are invisible in the final document may be used as templates, alignment marks, and so forth while using Adobe Illustrator to edit a document.

– **n**

The **n** operator is similar to the **N** operator, but first closes the current path.

– F

The **F** operator fills the area enclosed by the current path with the current filling color or pattern, leaving it as an open path. The inside of the current path is determined by the zero winding rule. See *PostScript Language Reference Manual, Second Edition* for “insideness” testing by the PostScript interpreter.

– f

The **f** operator is similar to the **F** operator, but first closes the current path.

– S

The **S** operator strokes the current path with a line using the current stroking color or pattern. The line width is specified by the graphics state (see the **w** operator) and the line is centered on the path with its sides parallel to the path. The joins between path segments are specified by the line join parameter in the graphics state (see the **j** operator), the ends of the path segments or dash lines within a segment (see the **d** operator) are specified by the end cap parameter of the graphics state (see the **J** operator).

– s

The **s** operator is similar to the **S** operator, but first closes the current path.

– B

The **B** operator is similar to the **F** operator, but both fills and strokes the path, leaving it as open.

– b

The **b** operator is also similar to the **f** operator, but both fills and strokes the path, leaving it as closed.

6.8 Compound Paths

Compound paths are equivalent to PostScript language paths; Illustrator paths are somewhat simpler. Letter shapes are often compound paths—for example, the letter A, because it has the enclosed counter.

The syntax for compound paths is:

```
<compound path> ::=  <*u>
                     <compound path element>+
                     <*U>

<compound path element> ::=
                     <path object>|<compound group>

<compound group> ::=
                     <u>
                     <compound path element>+
                     <U>
```

6.9 Clipping (Masking) Operators

Masks are conceptually similar to *ruby lith* and similar products in the graphic art industry—a red surface that blocks sections of a photo or other art work. An object used as a mask and the objects that it masks are bounded in the file by the **q** and **Q** operators as though they were grouped. Masks can be made from a path, a group of objects, or one or more compound objects. Each new mask intersects the current clipping path in the same way that the **clip** operator does.

```
<group with a mask> ::=
                     <q>
                     {<object>}*
                     {<masked object>}*
                     <Q>

<masked object> ::=  <mask>|<object>

<mask> ::=           <path mask>|<compound path mask>

<compound path mask> ::=
                     <*u>
                     <compound path mask element>+
                     <*U>

<compound path mask element> ::=
                     <path mask>|<compound mask group>

<compound mask group> ::=
                     <compound mask bottom group>|
                     <compound mask non-bottom group>
```

```
<compound mask bottom group> ::=
                    {<A>}
                    <q>
                    <path mask>+
                    <Q>

<compound mask non-bottom group> ::=
                    {<A>}
                    <u>
                    <compound mask group>+
                    <U>
```

– q

The **q** operator is similar to the **u** operator, except that an object in the group specifies a *mask* (clip path). The mask is a boundary for subsequent objects in the group, so that only objects within the boundary are visible when the illustration is rendered.

– **Q**

The **Q** operator is similar to the **U** operator, except that it marks the end of a sequence of elements containing a mask. A **q** operator must precede it.

– **H**

The **H** operator neither fills nor strokes the current path, but does not consume the path. This operator is used when establishing a mask.

– **h**

The **h** operator is similar to the **H** operator, but first closes the current path. This operator is used when establishing a mask.

– **W**

The **W** operator intersects the current clip path in the graphics state with the current path and sets a new, reduced clip path in the graphics state. No marks are made outside the area enclosed by the current clip path by subsequent fill and stroke operations until a **Q** operator appears. The **Q** operator restores the masking that was in effect at the matching **q** operator. This operator is used to establish a mask.

6.10 Text as Masks

You can use text objects as masks by using the **Tr** operator with a *rendermode* of 4 through 7. See section section 7.5, “Text Rendering Operators” for a full explanation of text rendering. When using text objects as masks, the entire text object becomes the mask. Illustrator cannot preview this kind of masking on the screen, although it prints properly. To see the effect of text used as a mask on the screen, convert the text to outlines.

7 Text

Illustrator writes out two kinds of information about text: *revisable* information and *final-form* printing information. Revisable information consists of those settings the user makes and can examine in Illustrator’s various dialog boxes—font size, for example. Final-form information is regenerated by Illustrator once the file has been read—it primarily helps to print text properly. Final-form information consists of such information as character positioning. This section differentiates between revisable, required information and final-form information.

Note *You can safely leave final-form information out of files and still have Illustrator read them correctly and regenerate the information; however, the file itself will not be an official EPS file and will not print.*

There are three kinds of text in Illustrator 3.x.

- *Point text* is created by clicking the text tool to create an insertion point, then typing. The first line of the text block begins at the insertion point. A carriage return determines the line break.

- *Area text* is created by clicking the area text tool and dragging a rectangle, and then starting to type. You can also create area text by clicking a path, selecting the area text tool, and then typing. A single text object can contain multiple text area elements.

- *Text on a path* follows the shape of a path. It is created by clicking an *open* path.

Text can *overflow* an area frame and thus not be visible on screen or when printed. Illustrator 3.x still saves such overflow text to the file, however. *Linked areas* allow text to flow from one area to another. Linked areas are part of the same text object.

Note *You cannot link point text to an area text object. All flowed text must appear in area texts.*

Area text objects can also be grouped with paths laying on top so that the text in the text objects “wraps” around (or avoids) the paths. The syntax for wraparound text is

```
<wraparound group> ::=
                        <*w>
                        {<object>}*
                        {<wraparound objects>}*
                        <*W>

<wraparound objects> ::=
                        <text object>|<object>
```

Wraparound text begins with the ***w** operator. Zero or more objects follow, then a standard Illustrator text object follows, and is in turn followed by zero or more Illustrator graphic objects around which the text will wrap. The wraparound text group ends with the ***W** operator.

Text can be *invisible* if you set its style to *neither* fill nor stroke. The PostScript interpreter does not render such text on the page. However, Illustrator writes the text to the file with the same structure as if the text were rendered.

7.1 Text Syntax

The syntax of text in Illustrator 3.x is as follows:

```
<text object> ::=       <To>
                        <text at a point>|
                        <text area>|
                        <text along a path>
                        <TO>

<text at a point> ::=   <Tp>
                        <TP>
                        <text run>+

<text area> ::=         <text area element>+
                        {<overflow text>}

<text area element> ::=
                        <Tp>
                        <path object>
                        <TP>
                        <text run>+

<text along a path> ::=
                        <Tp>
                        <path object>
                        <TP>
                        <text run>+
                        {<overflow text>}*
```

```
<text run> ::=          {<text style>|
                        <paint style>|
                        <text position>|
                        <Tk>}*
                        <text body>

<text style> ::=        <Tr>|(render mode)
                        <Tf>|(font & size)
                        <Ts>|(rise and fall)
                        <Tz>|(horizontal scaling)
                        <Tt>|(tracking)
                        <TA>|(automatic kerning)
                        <TC>|(intercharacter spacing)
                        <TW>|(interword spacing)
                        <Ti>|(indents)
                        <Ta>|(alignment)
                        <Tq>|(hanging quotations)
                        <Tl>(leading)

<text position> ::=     (printing only)
                        <Tc>|(computed intechar spacing)
                        <Tw>|(computed interword space)
                        <Tm>|(text matrix)
                        <Td>|(translate)
                        <T*>|(translate down)
                        <TR>(reset matrix; found only in
                        pattern prototypes)

<text body> ::=         <Tx>|<Tj>|<T+>|<T–>

<overflow text> ::=     {<text style>|<paint style>|<TK>}*
                        <TX>|<T+>
```

7.2 Text Attributes

A *text attribute* is a quality of the text such as font, justification, stroke, or fill. There are three kinds of text attribute used in Adobe Illustrator 3.x: character style, paragraph style, and paint style.

A *character attribute* is an attribute such as font size or scale that pertains to one or more characters. A*paragraph attribute* pertains to one or more paragraphs, and includes details such as justification and indentation. A *paint style attribute* pertains to any set of characters that all have the same paint style and character attributes (called a *text run*). You can apply any kind of paint style to characters—stroke or fill, stroke width changes, dashed lines, and so forth. In Illustrator, you set character and paragraph attributes in the Type Style dialog box.

7.3 Text Object Operators

A text object is bracketed by the **To** and the **TO** operators. The **To** operator begins a text object, and the **TO** operator ends it. Between the two can appear a series of operators that control the text path(s), set the text matrix, and set various other text attributes. Attributes are *modal*, that is, once an attribute operator has made a change, that change stays in force until changed again. While there is no special order in which the operators must appear, because of modality, operator order may have a bearing on the end result. See section section 7.1, "Text Syntax" for information about operator order.

A *text path* is a combination of the current transformation matrix and the path geometry of a *text container*—an area or object such as a box, circle, or other shape.

Note *If the Illustrator file includes more than one path between the* **To** *and* **TO** *delimiters, it must be area text; it means that text can flow from one object to another.*

7.4 Text Path Operators

Within a text object, each text path is bracketed by the **Tp** and **TP** operators. The **Tp** operator takes as its arguments a transformation matrix and a start point. The **Tp** and **TP** operators appear in every kind of text object. Its purpose is to provide a matrix for the text object and its container (if any). The operators vary in their use depending on text type.

- *Point text.* There is no path geometry. The **Tp** operator simply passes the matrix that is associated with the object.

```
<text at a point> ::=     <Tp>
                          <TP>
                          <text run>+
```

```
1 0 0 1 116 527 0 Tp
TP
```

- *Text on a path*. There is no path geometry for the text. The entire text path is rotated (skewed, etc.). The actual path is bracketed inside **Tp** and **TP**. The *startPt* parameter is used only for text on a path. It indicates where the text starts on the path by giving the fractional number of the Bézier segment it starts on begining with 0. For example, text that starts in the center of the second Bézier segment would have a *startPt* of 1.5.

```
<text along a path> ::=
                    <Tp>
                    <path object>
                    <TP>
                    <text run>+
                    {<overflow text>}*
```

```
1 0 0 1 114 479 0 Tp
243 403 m
243 479 L
114 479 L
114 403 L
243 403 L
n
TP
0 -10.875 Td
0 Tr
0 O
0 g
(The quick brown fox ) Tx
0 -14.5 Td
(jumps over the lazy ) Tx
0 -14.5 Td
(dog.  The quick brown ) Tx
0 -14.5 Td
(fox jumps over the lazy ) Tx
0 -14.5 Td
(dog.  ) Tx
(\r) TX
```

- *Area text*. Each container that text flows into has its own **Tp** and **TP** pair and matrix associated with it. The container also includes information about stroke and fill for the container.

```
<text area> ::=     <text area element>+
                    {<overflow text>}

<text area element> ::=
                    <Tp>
                    <path object>
                    <TP>
                    <text run>+
```

```
1 0 0 1 -342 1044 1.5104 Tp
109.5 309.5 m
117.8551 317.4372 116.4004 328.658 119.8602 338.3717 c
122.5416 345.9002 128.599 351.0478 134.564 356.4291 c
161.757 380.9608 206.5173 380.4135 231.1035 353.1425 c
246.4752 336.0923 253.1938 311.7104 275.641 302.8574 c
318.7067 285.8727 367.8791 298.6578 406.5 321.5 C
406.1633 321.5 405.835 321.5 405.5 321.5 c
```

```
N
TP
0.7152 0.6989 -0.6989 0.7152 126.0945 348.4862 Tm
0 Tr
0 O
0 g
(T) Tx
0.7435 0.6687 -0.6687 0.7435 131.317 353.508 Tm
(h) Tx
0.795 0.6066 -0.6066 0.795 136.1212 357.8995 Tm
(e) Tx
0.8733 0.4871 -0.4871 0.8733 143.9726 363.6125 Tm
(q) Tx
0.9148 0.4038 -0.4038 0.9148 149.6282 366.7758 Tm
(u) Tx
0.9388 0.3444 -0.3444 0.9388 155.631 369.3596 Tm
(i) Tx
0.9575 0.2886 -0.2886 0.9575 158.0023 370.2853 Tm
(c) Tx
0.9776 0.2105 -0.2105 0.9776 163.5868 371.975 Tm
(k) Tx
0.9965 0.0829 -0.0829 0.9965 172.4898 373.7108 Tm
(b) Tx
0.9999 0.0118 -0.0118 0.9999 178.9926 374.2097 Tm
(r) Tx
0.9982 -0.0598 0.0598 0.9982 182.8403 374.3088 Tm
(o) Tx
0.9865 -0.1638 0.1638 0.9865 189.295 373.9865 Tm
(w) Tx
0.9633 -0.2685 0.2685 0.9633 197.6105 372.5656 Tm
(n) Tx
0.924 -0.3824 0.3824 0.924 206.9198 369.6766 Tm
(f) Tx
0.893 -0.4501 0.4501 0.893 209.8895 368.5214 Tm
(o) Tx
0.8455 -0.5339 0.5339 0.8455 215.6646 365.6044 Tm
(x) Tx
0.776 -0.6307 0.6307 0.776 223.1863 360.5216 Tm
(j) Tx
0.7277 -0.6859 0.6859 0.7277 225.1888 358.9963 Tm
(u) Tx
0.6375 -0.7705 0.7705 0.6375 229.9913 354.5899 Tm
(m) Tx
0.5733 -0.8194 0.8194 0.5733 236.1842 346.9666 Tm
(p) Tx
0.5446 -0.8387 0.8387 0.5446 239.996 341.486 Tm
(s) Tx
0.5393 -0.8421 0.8421 0.5393 245.0267 333.6389 Tm
(o) Tx
0.5628 -0.8266 0.8266 0.5628 248.7057 327.8809 Tm
(v) Tx
0.6097 -0.7926 0.7926 0.6097 252.2511 322.6429 Tm
(e) Tx
0.6677 -0.7444 0.7444 0.6677 256.6117 317.0635 Tm
(r) Tx
0.7644 -0.6448 0.6448 0.7644 262.2055 311.2463 Tm
(t) Tx
0.838 -0.5456 0.5456 0.838 265.22 308.6137 Tm
```

```
(h) Tx
0.9226 -0.3857 0.3857 0.9226 271.7556 304.4719 Tm
(e) Tx
0.9542 -0.2992 0.2992 0.9542 281.684 300.6715 Tm
(l) Tx
0.9667 -0.2561 0.2561 0.9667 284.4584 299.7607 Tm
(a) Tx
0.9802 -0.198 0.198 0.9802 291.2396 297.9861 Tm
(z) Tx
0.9896 -0.144 0.144 0.9896 297.4289 296.7456 Tm
(y) Tx
0.9982 -0.0602 0.0602 0.9982 307.1617 295.4818 Tm
(d) Tx
1 -0.0054 0.0054 1 314.1365 295.0711 Tm
(o) Tx
0.9989 0.0468 -0.0468 0.9989 321.1079 295.0431 Tm
(g) Tx
0.9964 0.0842 -0.0842 0.9964 327.9832 295.4014 Tm
(.) Tx
(The quick brown fox jumps over the lazy dog.\r) TX
```

A text area element within a text object may be arbitrarily scaled, skewed, rotated, and translated using matrix operators. Text attributes such as whether the text is filled or stroked, its kerning and leading are controlled by attribute operators.

Adobe Illustrator has three text matrix handling operators that control where to place the text object. For text on a path, the **Tm** operator sets the text matrix. The **Td** and **T*** operators translate the text matrix to the start of the next line of text.

However, the three text matrix operators are final-form. Illustrator writes this information when it saves a file but ignores it when reading the file.

7.5 Text Rendering Operators

Text rendering mode is set by a call to the **Tr** operator. This operator takes as its argument one of the selectors in Table 1. The **Tx** and **Tj** operators actually render the text.

Table 1 *Text rendering modes*

Selector	*Rendering Mode*
0	fill text
1	stroke text
2	fill and stroke text
3	text with no fill and no stroke ("invisible")
4	mask and fill text
5	mask and stroke text
6	mask, fill, and stroke
7	mask (only) text
8	filled text followed by rendertype 9 (pattern prototype only)
9	stroked text preceded by render mode 8 text (pattern prototype only)

7.6 Kerning

Illustrator 3.x offers two types of kerning. These are:

- *Track kerning* (**Tt**). The specified amount of space is added between each pair of characters. Space is measured in thousandths of an em.

- *Pairwise kerning* (**TA**, **Tk**, **TK**). The **TA** (automatic kerning) operator tells Illustrator to turn automatic kerning on and use the kerning pairs specified in the font program itself. The **Tk** operator causes a manual or automatic kern—Illustrator puts a **Tk** in the file where kerning is necessary. Manual kerning overrides automatic kerning. Measurements that come from the kerning dialog box are expressed in thousandths of an em. The **TK** operator is identical to the **Tk** operator, but applies to overflow text that is not displayed or printed.

7.7 Spacing

Spacing control operators apply to justified and non-justified text. In the case of non-justified text, only the optimum value is used.

- *Word spacing* (**TW**). Specifies the minimum, optimum, and maximum space between words (space characters). Measurement is in a percentage of the width of a regular space character.

- *Character spacing* (**TC**). Specifies the minimum, optimum, and maximum space between characters in the file. Measurement is in a percentage of the width of a regular space character.

Two other spacing operators also appear in Illustrator files. They are final-form.

- *Computed word spacing* (**Tw**). This is the actual word spacing for a particular line of text. It is recomputed on a per-line basis.

- *Computed character spacing* (**Tc**). This is the actual character spacing for a particular line of text. It is recomputed on a per-line basis.

7.8 Line Spacing and Discretionary Hyphens

Line and paragraph leading is set using the **Tl** (lowercase L) operator, in units of one thousandth of an em-square. Superscripting and subscripting are set using the **Ts** operator with (respectively) positive and negative values expressed in points.

The **T+** operator appears whenever the user has inserted a discretionary hyphen.

The **T-** operator appears in text wherever a discretionary hyphen prints.

7.9 Alignment and Justification

A group of characters that share the same size, paint style, tracking, and so forth can be written out as a single call to **Tx** (for non-justified text) or **Tj** (for justified text).

The **Ta** operator sets text alignment: left, centered, right, or justified.

The **Ti** (lowercase i) operator controls line indentation. It takes arguments that state the values of the left indent, a delta from the left indent for the first line, and a right indent in points.

The **Tz** operator condenses or expands the horizontal scaling of a character as a percentage of the regular font size.

7.10 Text Operators

The following list of text operators includes whether the operator is revisable or final-form. **R** signifies a revisable operator, **F** signifies a final-form operator. Illustrator recalculates the values of all such final-form operators after reading the file.

type **To**

[Revisable]. This operator begins a text object. The type argument can be one of:

0	point text
1	area text
2	path text

TO

[Revisable]. The **TO** operator ends a text object and restores the current transformation matrix.

[a b c d tx ty] startPt **Tp**

[Revisable]. This operator brackets the text path. It concatenates the matrix parameter with the current transformation matrix (CTM). For text on a path only, the *startPt* operand indicates where the text starts on the arc length of the path by giving a fractional number that signifies its position along the Bézier segment on which the path starts, begining with 0. For example, text that starts in the center of the second Bézier segment would have a *startPt* of 1.5.

TP

[Revisable]. The **TP** operator ends the text path.

Matrix Operators

[a b c d tx ty] **Tm**

[Final-Form]. The **Tm** operator sets the text matrix for text along a path.

tx ty **Td**

[Final-Form]. This operator translates the text matrix by t_x and t_y. to the beginning of the next line of text.

T*

[Final-Form]. The **T*** operator translates the text matrix by *–lineleading, 0* to the beginning of the next line of text.

[a b c d tx ty] **TR**

[Final-Form]. The **TR** operator resets the pattern matrix for the pattern prototype only. See section section 5.4, “Pattern Definition” for more information about patterns.

Text Attribute Operators

render **Tr**

[Revisable]. The **Tr** operator sets the render mode for any text that follows. The **Tx** and **Tj** operators actually render the text. The *render* argument can be one of the following values:—

0—fill text
1—stroke text
2—fill and stroke text
3—invisible text
4—mask and fill text
5—mask and stroke text
6—mask, fill, and stroke text
7—mask (only) text
8—filled text followed by render mode 9 (pattern prototype only)
9—stroked text (preceded by render mode 8 text, pattern prototype only)

Modes 4 through 7 are used within pattern definitions when text, as part of the pattern, is filled *and* stroked with two different colors. Modes 8 and 9 are used in pattern prototypes that contain text objects.

fontname size **Tf**

[Revisable]. The **Tf** operator specifies the reencoded name of the font to use and its size in points.

alignment **Ta**

[Revisable]. The **Ta** operator sets text alignment both horizontally and vertically. The value for *alignment* can be one of the following:

0—left aligned
1—center aligned
2—right aligned
3—justified (right and left)
4—justified including last line

leading paragraphLeading **Tl**

[Revisable]. The **Tl** (lowercase l) operator sets the leading for the paragraph. The *leading* argument sets the leading between lines within a paragraph and the *paragraphLeading* argument sets the extra leading between paragraphs.

userTracking **Tt**

[Revisable]. The **Tt** operator sets additional space to add between characters in units of one thousandth of an em.

minSpace optSpace maxSpace **TW**

[Revisable]. The **TW** operator sets word spacing, where the minimum, optimum, and maximum space between words (the size of space characters) is expressed as a percentage of the width of a regular space character: 100% equals the width of one space character.

wordSpace **Tw**

[Final-Form]. The **Tw** operator sets computed word spacing.

minSpace optSpace maxSpace **TC**

[Revisable]. The **TC** operator sets character spacing, where the minimum, optimum, and maximum space between characters is expressed as a percentage of the width of a regular space character: 100% equals the width of one space character.

charSpace **Tc**

[Final-Form]. The **Tc** operator sets computed char spacing.

rise **Ts**

[Revisable]. The **Ts** operator sets the distance by which a character is raised above the baseline in superscripting and dropped below the baseline when subscripting. The argument *rise* is expressed in points.

firstStartIndent otherStartIndent stopIndent **Ti**

[Revisable]. The **Ti** operator sets the indentation of a paragraph. The argument *firstStartIndent* specifies the indentation for the left side of the paragraph, *otherStartIndent* specifies a delta from the left side that applies only to the first line, and *stopIndent* specifies any right side indentation. The three arguments are specified in points.

scalePercent **Tz**

[Revisable]. The **Tz** operator sets horizontal scaling of a line of text by condensing or expanding the line in terms of a percentage of the normal font width.

autoKern **TA**

[Revisable]. The **TA** operator specifies whether to use pairwise kerning. Illustrator uses the kerning pairs built into the Type 1 font program. If the *autoKern* argument is 0, Illustrator uses no pair kerning; if the *autoKern* argument is 1, Illustrator does use pair kerning.

hangingQuotes **Tq**

[Revisable]. The **Tq** operator determines whether a run of text uses hanging quotation marks; a hanging quotation mark is one that extends beyond the left or right edge of the text block. If the *hangingQuotes* argument is 0, Illustrator does not use hanging quotation marks; if the argument is 1, it does. For the domestic version of Adobe Illustrator, the marks which hang are: period, comma, semicolon, colon, backquote, left and right single quotes, left and right double quotes, dash, en-dash, em-dash. That is:

. , ; : ` ‘ ’ “ ” - – —

Note *Marks may be different for international editions.*

Text Body Operators

textString **Tx**

[Revisable]. The **Tx** operator holds a string to be used as the text of a path object. The string is not justified. The *textString* argument is any series of ASCII codes in the native platform encoding; the character combination \r signifies a new paragraph.

textString **Tj**

[Revisable]. The **Tj** operator renders a string of justified text. The *textString* argument is any series of ASCII codes.

textString **TX**

[Revisable]. Text that overflows the text area (invisible). Equivalent to the **Tx** operator.

autoKern kernValue **Tk**

[Revisable]. The **Tk** operator indicates that kerning takes place. A value of 0 or 1 for *autoKern* tells Illustrator whether to use manual (0) or automatic (1) kerning. The *kernValue* operand sets a kerning value in units of one thousandth of an em. When *autoKern* is 0, Illustrator uses the value in *kernValue*; when *autoKern* is 1, Illustrator uses the built-in kerning pairs.

autoKern kernValue **TK**

[Revisable] The **TK** operator is the same as the **Tk** operator, except that it applies to overflow text (invisible).

T+

[Revisable]. The **T+** operator marks every instance of a discretionary hyphen.

T-

[Revisable]. The **T-** operator marks where a discretionary hyphen has been printed.

8 Guide Operators

Guides act as control lines or shapes, and are similar to the “grid” feature in some drawing programs. They do not print and do not show up during preview, but other objects “snap to” guides for positioning. Guides can be lines or objects.

When you turn an object into a guide, it retains its color and other attribute information. If and when you release the object from being a guide, its attributes are restored.

Adobe Illustrator 3.x writes guides out to the file in the form of a path. The ***** operator begins and ends a path used as a guide. The ***** operator takes a string parameter which is one of the path render operators, for example **(F)***. The default form for guides created via the ruler is **(N)***. For example:

```
(N) *
   315 1044 m
   315 -252 L
   (N) *
```

Guides can appear within groups, and guides themselves can be groups.

9 Placed Art Operators

The syntax for placing an EPS file into an Adobe Illustrator document is

```
<placed art object> ::=
                        <'>
                        <art reference>
                        <~>

<art reference> ::=     <file reference>|<file inline>

<file reference> ::=    %%IncludeFile: <filename>

<file inline> ::=       %%BeginDocument:<filename>
                        … included file contents
                        %%EndDocument

<filename> ::=          platform-specific path name of file

<subscriber object> ::=
                        %AI3_Subscriber:<subscriber ID>
                        <placed art object>

<subscriber ID> ::=     resource number of SECT resource in file
```

The **%%subscriber** comment concerns the Macintosh publish and subscribe facility available in System 7. It reads in an edition which has been published as graphics. The *section ID* shows the resource ID of the section (in a *sect* resource), as stored in the file. See *Inside Macintosh Volume VI* for more information on publish and subscribe. If you need to use this feature, you will also need to store the bounds in a *bnds* resource of the given id (whose format is a rectangle of Fixed values: left, top, right, bottom), and the section options in a *psop* resource of the given id (whose format is a two-byte integer, value 0).

The filename in the **%%IncludeFile** comment must be the same as that specified for the **'** (single tick-mark) operator. This includes a file *by reference* in a given Illustrator document.

You can also save included files *directly* (rather than by reference) in an Adobe Illustrator document. If you wish to do that, replace the **%%IncludeFile** comment with the structure:

```
%%BeginDocument: filename
   … put included file here
   %%EndDocument
```

Adobe Illustrator automatically modifies the **%%DocumentFonts** comment of the including document to add any fonts in the **%%DocumentFonts** comment of an included document.

[a b c d t_x t_y] llx lly urx ury (filename) **‘**

The **‘** (single quote or tick-mark) operator specifies that the document stored in *filename* is to be imported into the illustration. The imported file is assumed to be an EPS-conforming document.

The *filename* string is the full pathname for the file in the operating system’s file system. Adobe Illustrator concatenates the matrix operand with the current transformation matrix to establish a new user space and an origin for the imported document. See *PostScript Language Reference Manual, Second Edition* for more information about transformation matrices. The matrix handles any rotation and reflection to be applied to the imported document. The *llx lly urx ury* operands specify the bounding box from the imported document as stated by the **%%BoundingBox** comment in the imported document’s prolog. In addition to establishing a new user space, the **‘** operator does the following (or their equivalents):

```
false setoverprint
   0 setgray
   0 setlinecap
   1 setlinewidth
   0 setlinejoin
   10 setmiterlimit
   [ ] 0 setdash
   newpath
```

– **~**

The **~** operator restores the user space in effect when the preceding **‘** operator was executed.

10 Graphs

Illustrator’s *graphs* capability builds business graphs, such as bar (column) graphs and pie charts. You can build such graphs using Illustrator’s general drawing abilities, but having Illustrator itself build the graphs from data you enter or import helps assure accuracy.

The **Gs** operator begins a graph and the **GS** operator ends it. All graph operators consist of two characters and begin with **G**.

A graph in an Illustrator file has two parts: the *functional specification* and the *objects* that make up the graph. The functional spec acts like an internal header; it contains information about the graph, such as graph type, axis parameters, and graph data values. The graph objects are regular Illustrator path and text objects, which allows the file to be incorporated as an EPS file in other files for documents.

On command, Illustrator can recreate a graph based only on the functional spec. However, on reading an Illustrator file that contains a graph, Illustrator *does not* automatically recalculate the graph. Instead, it displays the graph objects as they appear in the file—exactly as Illustrator would treat any other objects in a file.

Note *When preparing files that contain graphs for use with Illustrator, you must make sure that the functional spec matches the graph objects (if you choose to supply them). Otherwise, recalculating will result in changes to the graph.*

10.1 Parts of a Graph

holds labeled illustration
Those parts are:

axis
label group
axis tick group
category axis group
edge
legend group
data column
series 0
series 1

10.2 Syntax

The syntax of the graph section is:

```
<graph object> ::=      <Gs>
                        <graph functional spec>
                        {<graph customizations>}
                        <graph group object>
                        <GS>

<graph functional spec> ::=
                        <graph size and dialog values>
                        {<graph subscriptions>
                        <graph axis>
                        <graph axis>
                        <graph axis>
                        <graph table specs>

<graph size and dialog values> ::=
                        <Gb>
                        <Gy>
                        <Gd>
```

```
<graph axis> ::=          <Ga>
                          <GA>

<graph table specs> ::=
                          {<Gw>}*
                          <Gz>
                          <Gc>+
                          <GC>

<graph customizations> ::=
                          <Gt>
                          {<graph customization>}*
                          <GT>

<graph customization> ::=
                          {<graph customization operator>}*
                          <GX>|<Gg>
                          {<Gv>}

<graph customization operator> ::=
                          {<Gm>}
                          {<Gf>}
                          {<Gy>}
                          {<GD>}
                          {<Ge>}
                          {<G1>}(gee-one)
                          {<Gi>}
                          {<Gl>}(gee-ell)
                          {<Gp>}
                          {<Gx>}
                          {Gr>}
                          {<G+>}
                          {<Gg>}
                          {<A>}
                          {<paint style>}*
                          {<text style>}

<graph subscriptions> ::=
                          <Gj>

<graph group object> ::=
                          <u>
                          {<graph rendered object>}*
                          <U>

<graph rendered object> ::=
                          <object>|<graph group object>
                          {<Go>}
```

10.3 Functional Specification

The functional specification acts as an internal header. It contains information about the bounds of the graph, its style, its axes, and the data that make it up. From the functional spec, Illustrator can reconstruct the graph (Illustrator does not automatically recalculate the graph on reading in a file); in fact, Illustrator can reconstruct a graph from no more than the functional spec and does not require graph objects at all.

Statements in the functional spec take the form of comments, and begin with the characters **%_**. This allows an Illustrator file with a graph in it to be used as an EPS file, in which comments are not executed. The following example shows a simple functional spec.

```
%_Gs
%_548 122 301 440 Gb
%_5 0 0 14 1 1 0 0 3 44 Gy
%_7 2 1 0 90 80 0 Gd
%_1 () Ga 0 14 0 1 0.2 0 1 () GA
%_2 () Ga 0 14 0 5 1 0 0 () GA
%_4 () Ga 0 14 0 5 1 0 0 () GA
%_4 4 1 1 Gz
%_() Gc (FIRSTCOL) Gc (SECONDCOL) Gc (THIRDCOL) Gc (GLOVES) Gc 1 3
   1 (BALLS) Gc 2 2 5 (BATS) Gc 3 4 2 Gc GC
```

The **%_Gs** operator opens the graph section. The next operator, **Gb**, defines the bounds of the graph and determines where Illustrator draws the graph's axes. **Gy** and **Gd** apply some of the values from Illustrator's Graph Style dialog box. The three **GA** operators control how values appear on the axes. The **Gz** operator works with the **GA** operator to specify the graph axis.

The **Gc** operator reads in cell values, one by one, and puts them in the cell data table—rows and columns like a spreadsheet—from which Illustrator constructs its graphs. Operators in Illustrator are limited to one string parameter and a maximum of 16 parameters overall; consequently, there is usually more than one **Gc** operator to a line. The cells in the series read from left to right in a row; then the next row down is read in. String values appear within parentheses. If a cell holds no value, the file holds its place with a pair of empty parentheses. In the example

```
%_() Gc (FIRSTCOL) Gc (SECONDCOL) Gc (THIRDCOL) Gc (GLOVES) Gc 1 3
   1 (BALLS) Gc 2 2 5 (BATS) Gc 3 4 2 Gc GC
```

the contents of row1 column1 is an empty string. Because it must be allowed to hold a string, the **Gc** operator follows it (one string allowed per operator). The contents of row1 column2 is the string *FIRSTCOL*. Again, because it is a string, the **Gc** operator must immediately follow it. The **Gc** parameter doesn't "care" about the coordinates of the parameters it sets, only their number. For example, if there were 40 numerical data points to place in the table, the first **Gc** can place 16, the second **Gc** can place 16, and the third **Gc** can place eight—with no regard to where those 40 data points fall on the table. It is the *order* in which they occur within the file that determines their row and column coordinates.

If you are preparing a file to be used with Illustrator, label and data point order is critical to creating an accurate graph.

The final **GC** ends the data table.

Note *Illustrator notes any edits you make to the graph after you build it initially; it specifies those edits as a series of changes following the functional spec. These customizations are covered in section section 10.5."*

10.4 Operators in the Functional Spec

Gs

This operator signals the beginning the graph object. Must be the first operator in this graph object. It takes no parameters.

GS

This operator signals the end the graph object. It takes no parameters.

left top right bottom **Gb**

The **Gb** operator shows the *bounds* of the graph. The parameters are decimal numbers, in the same coordinate system as other objects. This rectangular area defines where Illustrator draws the axes' lines. The axis labels, category labels, legend boxes, and legend labels are all slightly outside this rectangle. The axis lines are drawn exactly on the edges of this rectangle. This is a pre-customization rectangle. See section section ."

graphType shadow dataPaintOrder pieLegendStyle drawMarks drawLines drawLinesAsShapes drawLegendsAcrossTop lineShapeWidth whichAxis [piePercentage piePctDigits] **Gy**

This operator defines values for the Graph Style dialog box, just as does **Gd**; however, these values can be applied to individual series in a graph, overriding, for that series, the default graph style.

graphType — This is the style of graph, for the whole graph (which may be overridden in any number of series), or for one series. The values for *graphType* are:

5—grouped column graph
6—stacked column graph
7—line graph
8—pie chart
9—scatter graph
10—area graph

Note — *The values for scatter and area are in the opposite order from the dialog box and tools.*

shadow — This controls whether to create an object showing a shadow when creating the graphical objects. 1 means draw the shadow. Values: 0 or 1. Default: 0.

dataPointOrder — When individual items in a data series overlap graphically, the items corresponding to the upper rows in the cell table are covered by the graphical objects corresponding to the lower rows in the cell table. A value of 1 does the opposite. See the *seriesPaintOrder* parameter in the **Gd** operator for more information.

pieLegendStyle — Legends in pie charts can be numerous types; they can be the same as for bar and line graphs (boxes along the right or top edge, with labels next to them), a label within each wedge, or none at all. The values are:

14	same as bar/line graphs
15	legends in wedges
16	no legends

drawMarks — When creating line and scatter graphs, Illustrator can place small marks at the data points or have lines go through them without special marks. This parameter controls the existence of the marks. 1 means draw marks. Values: 1 or 0. Default: 0.

drawLines — When creating line and scatter graphs, Illustrator can draw lines that connect the data points, or not. This parameter controls the existence of the lines. 1 means draw them. Note that if this and the above value are both zero, Illustrator draws nothing within the data area. Values: 1 or 0. Default: 1.

drawLinesAsShapes	When creating line and scatter graphs, Illustrator can use simple lines to represent the data, or use shapes that represent thick lines. 1 means draw the shapes. Values: 1 or 0. Default: 0. Note that if *drawLines* is 0, this parameter is ignored (but still must be supplied).
drawLegendsAcrossTop	When creating a graph that contains legends, Illustrator can either put the legends on the right edge of the graph, going down, or on top, going left-to-right. 1 means put them across the top. Values: 0 or 1. Note that this is a parameter that applies to the graph as a whole and cannot be applied to individual series (although a value, as a placeholder, must appear here). Default: 0.
lineShapeWidth	If *drawLinesAsShapes* is on, this is the width (in points) of the shape that is created. Values: 0.0 to 100.0. Default: 6.
whichAxis	This shows which axis (left or right) against which to measure the data. Often, graphs are created that show trends of two series that have very different ranges (for instance, net income in millions of dollars against share price under one hundred dollars). One axis can show values in the millions while the other shows the values under 100. The values are: 44 Use left axis 45 Use right axis
piePercentage	This is a flag that tells Illustrator to display a percentage number inside a pie wedge. It is available only for Adobe Illustrator for Windows Version 4.x. A value of 1 tells Illustrator to display the percentage; 0 tells it not to display the percentage.
piePctDigits	This value tells Illustrator the number of digits to place after the decimal point when displaying a percentage in a pie wedge. It is available only for Adobe Illustrator for Windows Version 4.x.

colWidth cellTableDecimalPrecision seriesPaintOrder useBothAxis barWidthPercentage groupWidthPercentage drawLinesEdgeToEdge **Gd**

This operator defines some of the values in the Graph Style dialog box. Each parameter is defined following. The values for this operator apply to the entire graph; they cannot be applied to individual series in a graph (a series corresponds to a column in the cell table).

colWidth — This is the default column width of the cell table for this graph, expressed as the number of characters that will fit in that column. Inidividual column widths can be overwritten by the **Gw** operator. Range: 3 to 20.

cellTableDecimalPrecision — This is the number of decimal places shown in the cell table. The values are kept at full precision; only the display of those values is affected. Range: 0 to 10.

seriesPaintOrder — When Illustrator creates a graph, it creates the objects for each series. By default, it puts the each subsequent series in front of the previous one: that is, if they overlap, graphical objects representing later series will display and print on top of earlier series. A value of 1 displays and print the series in the opposite order; that is, the first series will be frontmost in the graph and will paint over other series if they overlap. Note that this does not change the position of the series; only the painting order. Values: 0 or 1; default value 0.

useBothAxis — The user can choose either left, right, or both axes to display, and independently set the axis numbers and tick placement for each axis. A parameter value of 1 “copies” the values from one axis to the other. Which axis is copied depends on which axis has been chosen as the default for all series (both may have been chosen, in which case Illustrator ignores this parameter). If one axis has been chosen, the other axis is copied. Values: 0 or 1. Default: 0.

barWidthPercentage — This is the percentage of the width available for a bar that will be used for that bar. (In the Illustrator manual, vertical-bar graphs are called “column graphs.” Use of the term “bar” distinguishes the graphic rectangle that represents one piece of data in the cell table from a column of cells in the table). Illustrator divides the available width of a graph into portions depending on how many series there are and how many individual bars there are per series. For example, if this parameter value is 100, for grouped-column graphs there would be *no* space at all between individual bars in a group. The lower the percentage, the more space between individual bars. The higher, the less—values over 100, which are legal, causes bars to overlap. See the *dataPaintOrder* parameter of **Gy** for how to influence the paint order of individual overlapping bars. For stacked-column graphs, individual bars are all in the same stack—the *barWidthPercentage* and *groupWidthPercentage* are multiplied to obtain the real percentage to use for each stack. If both are 100, no space shows between stacks. One value of 64 and the other of 100, versus both values of 80, shows the same graph for stacked-column graphs but shows different group *and* bar spacing for grouped-column graphs. Range: 1.0 to 1000.0 (that is, one-hundredth of the available width to ten times the available width).

groupWidthPercentage	This is the percentage of the width available for a group (in grouped-column graphs) or a single stack (in stacked-column graphs) that will be used for that group or stack. See *barWidthPercentage*, above, for a fuller description. Range: 1.0 to 1000.0 (that is, one-hundredth of the available width to ten times the available width).
drawLinesEdgeToEdge	In line graphs, the user can decide whether to leave a little space between the left and right axes and the beginning and end of the data lines. Usually this is done to leave room for labels at the bottom of the graphs. This parameter defines whether or not to draw the lines right to the edges. Note that this widens the area available for each label along the bottom, because the same number of labels now have a slightly wider area to draw. 1 means draw all the way to the edge. Values: 0 or 1. Default: 0.

leftColumn topRow rightColumn rightRow sectionID **Gj**

This operator concerns the Macintosh publish and subscribe facility available in System 7. It reads in an edition which has been published as text. The bounds (0 is the first column) show how big the section is, although that is ignored: if the read-in section is larger or smaller than the bounds, they're re-set by Illustrator. The *sectionID* shows the resource ID of the section (in a *sect* resource), as stored in the file. See *Inside Macintosh Volume VI* for more information on publish and subscribe. If you need to use this, you will need to store the *sectionHandle* in the *sect* resource with the given id, the bounds in a *bnds* resource of the given id (whose format is a rectangle of Fixed values: left, top, right, bottom), and the section options in a *psop* resource of the given id (whose format is a two-byte integer, value 0).

whichAxis beforeString **Ga**

This is the beginning of the specification of a graph axis. It continues and ends with the **GA** operator.

whichAxis — This tells which axis to set to this specification. The values are:

1	Bottom axis
2	Left axis
4	Right axis

beforeString — This string is appended to the label next to each axis tick mark; a value of *(units)* shows as “100 units” rather than as just 100. These labels show the numerical representation of the tick, which can include a currency symbol or some other representation *before* the value. If the value of *beforeString* is $, the axis label becomes $100 instead of 100. Length: up to 9 characters. Note that any characters other than standard ASCII must be entered as an octal number with a slash (this is a PostScript standard), so £ and ¥ must be decoded into their ASCII values.

useManualValues tickMarkType minimumValue maximumValue betweenValue smallTicksPerValue draw-MarksBetweenLabels afterString **GA**

useManualValues	The user can choose to have Illustrator decide which axis values are appropriate given the range of the cell data, or the user can specify his own values (for the current axis, specified by the **Ga** operator). 1 means use manual values. Values: 0 or 1. Default: 0.
tickMarkType	The user can specify short tick marks, tick marks that stretch to the other edge of the graph, or none. The values are: 13—no tick marks 14—short tick marks 15—long tick marks
minimumValue	If the user has specified manual values for the axis, this parameter supplies the lower of the two values. Depending on the *betweenValue*, *minimumValue* may correspond to the topmost or bottommost tick mark. Range: any decimal number.
maximumValue	If the user has specified manual values for the axis, this parameter supplies the higher of the two values. Depending on the *betweenValue*, *maximumValue* may correspond to the topmost or bottommost tick mark. Range: any decimal number.
betweenValue	If the user has specified manual values for the axis, this parameter supplies the numerical difference between subsequent tick marks. If this number is positive, then *minimumValue* shows up on the bottom of the *y*-axis (on the left in the case of the *x*-axis). If this number is negative, then *maximumValue* shows up on the bottom of the *y*-axis (on the left in the case of the *x*-axis). Range: any decimal number.
smallTicksPerValue	The user may want a graph with tick marks every 10, but labels every 20. Illustrator uses *smallTicksPerValue* to divide *betweenValue* into smaller segments and display extra tick marks without displaying labels. Illustrator

	divides the betweenValue by this number, and zero is a valid result meaning none. This means that 0 and 1 do the same thing: produce no small ticks. Range: integers between 0 and 1000. Default: 0.
drawMarksBetweenLabels	This applies only to the *category axis*. Sometimes the user wants tick marks on the category axis to line up exactly with the labels (usually in the case of line and area graphs), and sometimes the user wants ticks between the labels (usually in the case of stacked-column and grouped-column graphs). If 1, the category axis draws tick marks between the labels. Values: 0 or 1. Default: 0.
afterString	As with *beforeString* in the **Ga** operator, this string is appended to the label next to each axis tick mark; a value of *(units)* shows as “100 units” rather than as just 100. Length: up to 9 characters. See *beforeString* for character value limitations.

rows columns firstDataRow firstDataColumn **Gz**

This shows the size of the *cellTable* in rows and columns (1 means 1 row or column). It also shows the row index and column index of the first row/column containing data (0 means first row). Only 1 row/column of labels is allowed, so *firstDataRow* and *firstDataColumn* can be only 0 or 1. The other parameters can be anything from 1 to whatever will fit in memory. There must be at least one row and one column in the table. This operator must be followed by however many **Gc** operators it takes to fill the table with data.

cellValue$_1$ *cellValue*$_2$ *cellValue*$_3$ … *cellValue*$_X$ **Gc**

This operator reads in cell values, one by one, and puts them in the table. Every cell in the table must be enumerated, even if it’s empty. Use as many invocations of **Gc** as necessary until the table is full or until all subsequent cells are empty. The first cell value goes to the top-left cell, subsequent cells along the top row are filled until the number of columns (as specified in the **Gz** operator) has been reached. Then the next row is filled from left to right.

The number of parameters to **Gc** doesn’t necessarily match the number of columns in the table; it is constrained by the following rules: only one parameter can be a string and there cannot be more than 15 parameters. This means that if the top row contains any labels, there will probably be many **Gc** invocations, each with one parameter; then for the data rows there will be fewer invocations, each with a greater number of parameters. String parameters can be any length up to 255 characters (the maximum line length for illustrator is 128 characters), and numerical values can be anything. Empty cells are denoted by a set of empty parentheses.

For example, the following table and **Gc** operator and its operands are equivalent:

	FIRSTCOLSECONDCOLTHIRDCOL
GLOVES	131
BALLS	225
BATS	342

```
  %_() Gc (FIRSTCOL) Gc (SECONDCOL) Gc (THIRDCOL) Gc (GLOVES) Gc 1 3
    1 (BALLS) Gc 2 2 5 (BATS) Gc 3 4 2 Gc GC
```

col $width_1$ $width_2$ $width_3$ … $width_X$ *numParams* **Gw**

This overrides the default column width (expressed as a number of characters that fit in the table) given in the **Gy** operator. The first parameter is the column for which to start setting widths (0 is the first column); the next parameters (there must be at least one, and can be up to 14) set the width of that column and the next *n*(<14). To set the width for more than 14 columns, use multiple invocations of **Gw**. If –1 is the column width, it is considered a placeholder default column width. The operand *numParams* is the number of parameters to the operator, including the col parameter.

GC

The cell table is finished. This is an indication to Illustrator that the temporary state and data structures it has set up to read in the cell table can be disposed.

End of the Functional Specification

This ends the functional specification of the graph. All the operators shown so far should be written out with every line beginning with the PostScript comment characters **%_** (<percent> <underbar>).

10.5 Graph Customizations

After Illustrator has created the functional specification (and before it writes any of the graph objects), it tracks any edits that the user may have made to the graph in the form of changes to that functional spec. Such changes are called *customizations*.

Much like Paint Style parameters, Illustrator maintains a “running” customization—only the changes to the customization appear in the file. The **Gt** and **GT** operators bracket the customizations.

Illustrator writes out the **GX** operator to add the customization to the list for the graph.

If a **GX** operator appears *before* another operator with that parameter has appeared—anywhere within the entire set of customizations—Illustrator assumes that the parameter holds the initial value. For some parameters, the initial value is not one of the legal values. This means that the operator which contains this parameter must *always* appear somewhere in the list of customizations before the **GX** for a customization for which this parameter is necessary.

For example, if **Gx** appears in a customization that has been defined to apply to a column, but the **Gr** operator has not appeared, it is assumed that the target column is 0 (zero) since that is the initial value. That is, there is an implicit 0 Gl (zero gee-ell) at the beginning of the list of customizations. This applies to *all* operators. Note that some, like **Gi**, have a default value that is *illegal*. This means that before the **GX** operator appears on a customization that applies to an axis, there *must* be a **Gi** operator.

As another example, the bar design type parameter's initial value is 0, but that's not one of the legal values. Thus, before the **GX** appears for a bar-design customization, the operator containing the bar design type—**GD**—*must* appear). For each graph in a file, the value is reset to its initial state; that is, the "running" customization is not carried from graph to graph.

Note that some operators take parameters that do not have a meaning in the particular context; in this case, the parameters are used to set structure fields in the customizations, but they are ignored.

There can be fields that necessarily do not want a value; there is a special construct for this: **--** (<hyphen><hyphen>). This is used when a customization sets, for instance, the paint style of a group but the user did not fill in the color of the group. In the case where the group contains objects of various strokes—and the user changes the Paint Style without wanting to change the objects' strokes—Illustrator must create a Paint Style that does not change the stroke style but *does* change whatever the user wants changed.The **--** parameter and the **w** operator are used as part of the construction of a "Set Paint Style" customization, to make sure that this customization will not reset the stroke width of a target that has previously had its width set by another "Set Paint Style" customization.

Customization Operators

version **Gt**

This marks the beginning of users' customizations to the graph.The current version is 2.

GT

This marks the end of the list of customizations.

target graphCustomization **Gx**

This sets up the main part of the customization, itemizing the target of the customization and the basic customization type. The parameters are:

target is an index into all possible pieces of a graph. This is the object to which the customization is applied. For each type of target, other parameters to other operators will have to be filled in. For instance, if the target is one entire axis (9), the **Gi** operator should precede the **GX** operator, denoting which axis is the target. Initial value: 0. Possible targets are:

0	entire graph
1	all series, including legends
2	one series, including legends
3	one series but not its legend
4	one data bar/line/wedge
5	all data marks
6	one series' and its legend's marks
7	one series' marks, but not legend's
8	one data line segment's mark
9	one axis, including text, ticks, line
10	category axis' main line
11	one axis' set of major tick marks
12	one axis' single major tick mark
13	one axis' set of tick labels
14	one axis' single tick label

15	all legends' text
16	one legend's text
17	one numerical axis' main line
18	one legend's box or line, not mark
19	one legend's mark
20	all labels along category axis
21	one label of category axis
22	entire "shadow" object
23	every tick of one axis
24	all minor (small) ticks of one axis
25	one minor (small) tick of one axis

graphCustomization

This parameter enumerates the type of customization. If this is an Illustrator Customization, it indicates the *illustratorCustomization* (see the **Gp** operator). Initial value: 0. The values are:

0	Illustrator Customization
1	Set Series' Graph Style
2	Set Column (Bar) Design
3	Set Mark Design

illustratorCustomization **Gp**

For customizations that involve general illustrator operators (see list just below), this operator enumerates the type of customization. This is ignored if the *graphCustomization* parameter to the **Gx** operator indicates that this customization is a graph-specific customization. Initial value: 0. Values (for *illustratorCustomization*) are:

0	Move/Shear/Rotate/Scale
1	Set Paint Style
9	Send To Front/Back
10	Set Character Style
11	Set Layout Style

changeMethod **G+**

Customizations usually reset properties of objects regardless of the previous value of the property. A few, however, involve adding or subtracting something from the previous property. There is an Illustrator operator to add two points to the type size; to mimic this customization, the **G+** operator indicates that the customization's values are to be added to the current values, rather than replacing the values. Initial value: 0. Values (for *changeMethod*) are:

0	Reset to new value
1	Add new value to previous value

Note *Adding or subtracting values doesn't apply to the values* in the customization itself. *It affects* how *those values will apply to objects when Illustrator recalculates the graph.*

sendToFront **G1** (gee-one)

For the Send-to-Front and Send-to-Back customizations, the parameter denotes front or back (since the *illustratorCustomization* parameter of the **Gp** operator is the same for both operators). Initial value: 0. Values are:

0	Send to back
1	Send to front

doFill doStroke fillColorStyle strokeColorStyle isAMask **Gf**

For the Set Paint Style customization, these parameter denote the settings of the fill and stroke radio buttons that are not covered by the standard Paint Style operators. For those path objects not in graphs, the operator that finishes the path object denotes the values. Since this operator does not create a path object, but merely denotes its properties, there needs to be operators specifically for these values. The parameters are:

doFill	Used to denote that a non-empty fill style has been selected. 1 means that the customization creates a fill (of style *fillStyle*, below). Initial value: 0. Values: 0 or 1.
doStroke	Same as *doFill*, except it applies to the stroking. Initial value: 0.
fillStyle	Denotes the style of filling for this customization. Initial value: 0. The values are:

0	black (or white)
1	process
2	pattern
3	custom color
4	blend (applies only to Adobe Illustrator for Windows Version 4.0)

strokeStyle	Denotes the style of stroking for this customization. The values are the same as for *fillStyle*. Initial value: 0.
isAMask	This object has had the Mask box checked in the Paint Style dialog. 1 means the object is a mask. Initial value: 0. Values: 0 or 1.

column **Gl** (gee-ell)

For targets that are a series or a data point within a series, this is the column index in the table that corresponds to the series. Note that 0 (zero) means the first column that contains graphable numeric data; that is, if the first column consists of labels, then the second column is considered zero. Initial value: 0.

row **Gr**

For targets that are a series or a data point within a series, this is the row index in the table that corresponds to the index within the series indicated by the **Gl** (gee-ell) operator. Note that 0 (zero) means the first row that contains graphable numeric data; that is, if the first row is labels, then the second row is considered zero. Initial value: 0.

whichAxis **Gi**

For a customization that applies to an object inside an axis, this operator tells Illustrator *which* axis the object is inside.

whichAxis	For targets that are part of an axis, this parameter denotes the axis (left, right, bottom or top). Initial value: 0.

The values are:

1	Bottom axis
2	Left axis
4	Right axis
8	Top axis (Adobe Illustrator Japanese Edition only)

a b c d h v generalGraphType reserved1 reserved2 **Gm**

For customizations that use matrices (the *graphCustomization* parameter in the **Gx** operator is Illustrator Customization, and the *illustratorCustomization* parameter in the **Gp** operator is Move/Shear/Rotate/Scale). The parameters are:

a, b, c, d, h, v	These are the values for the matrix. They (the *h* and *v*) are in artwork-coordinates, and all are decimal values. Initial values: 0, 0, 0, 0, 0, 0.
generalGraphType	Some customizations only look good for certain general types of graphs; that is, shearing of the bars in stacked-column graphs would not look good if applied to the data in pie charts, yet when applied to grouped-column graphs it achieves a good effect. This denotes the general type of graph to which the customization should be applied. This parameter will have meaning in the case where the user reads this file into Illustrator, then changes the graph type. Initial value: 0. The values are:

1	Grouped- and Stacked-column
2	Scatter and line graphs
3	Pie charts
4	Area graphs
5	All graphs

reserved1	Set to 0.
reserved2	Set to 0.

designName designType repeatPartialType rotateLegend **GD**

This operator sets up the parameters for bar design customizations (see the **Gp** operator).

designName	This is the name of the design. The design was set up, just like a pattern, in the prolog. Initial value: zero-length string.
designType	This denotes the type of the design. Initial value: 0. The values are:

6	Vertically-scaled design
7	Uniformly-scaled design
8	Repeating design
9	Sliding design

repeatPartialType — For repeating bar design customizations, this denotes whether to chop the design representing any partial values, or to scale it. Initial value: 0. The values are:

16	Chop partial values
17	Scale partial values

rotateLegend — When using bar designs, the user can have Illustrator automatically rotate the design in the legend box. This denotes whether to do it. 1 means rotate the design in the legend box. Initial value: 0. Values: 0 or 1.

repeatEachValue **Ge**

For customizations that set the bar design (see **Gp** operator), and for which the design type is a repeat design (see **GD** operator), this operator denotes the value for each repetition of the design. Initial value: 0.0. Values: any decimal number but zero.

tickValue **Gv**

For targets that are either one tick mark or one tick label, this is the numeric value corresponding to that tick mark. That is, if the user changes to red the tick line next to the 100 on the tick axis, this number tells Illustrator that any tick line on that axis that corresponds to the value 100 (no matter how many tick lines there are or what their values), is to be red. Initial value: 0.0. Values: any decimal number.

Note *This is the only customization operator that appears after the* **Gx** *operator within one customization.*

GX

This finalizes the customization, which has been described by any of the previous values.

target column row whichAxis illustratorCustomization graphCustomization ignored changeMethod **Gg**

Note *This operator is not written by Illustrator; all the parameters are redundant with parameters of other, more efficient, operators. While Illustrator does currently read this in, and some files written by beta versions of Illustrator 3.x may have this operator in it, it is highly recommended that the other operators be used instead. Future versions may not read this operator.*

This sets up the main part of the customization, itemizing the target of the customization and the basic customization type. The parameters are:

target	See “Gx” operator.
column	See “Gl” operator.
row	See “Gr” operator.
whichAxis	See “Gi” operator.
illustratorCustomization	See “Gp” operator.
graphCustomization	See “Gx” operator.
ignored	Unused. Set to 0 (zero).
changeMethod	See “G+” operator.

10.6 Graph Objects

Graph objects make up the second part of the graph. When Illustrator loads a file containing a graph, it draws the objects as described by this section without referring to the functional specification (at will, a user can tell Illustrator to recalculate the graph based on the functional spec).

Graph objects are essentially standard Illustrator objects, use Illustrator operators for their construction, and are subject to the current paint, fill, and stroke styles and the rest of the cumulative settings of the graphics state. However, in order to allow efficient editing of graphs, Illustrator uses a particular hierarchical grouping organization in memory. This has two effects on the file.

- Each graph object is followed by the **Go** operator, which tells Illustrator what that object is—data point, legend, axis, and so forth. See section section 10.1” for an illustrated look at the various parts of a graph.

- Graph objects are grouped in a way representing their hierarchy in memory using the **u** and **U** operators (just as with other Illustrator grouped objects). Because this grouping represents a data structure, there may be in the file one or more “empty” groups of **u** followed immediately by **U**; there may even be nested empty groups.

Note *When constructing an Illustrator file external to Illustrator which contains a graph, you must make sure that the organization of the empty groups is followed accurately.*

Organization of Graph Objects

The following table shows the hierarchical organization of graph objects. Each group is bounded by the **u** and **U** operators.

```
Left Axis group
Right Axis group
Legend group
Category Axis group
Data
        Series 0
             Legend box
             data series 0
        Series 1
             Legend box
             data series 1
        Series n
             Legend box
             data series n
Axis
```

Graph Object Operators

target column row whichAxis **Go**

After each object that represents a major part of the graph, Illustrator must identify it: which part of the graph did Illustrator just read in? These parameters specifiy that. This operator is not needed for any part of the graph that is not covered by the following parameters. Note that some of the parameters are not needed for several of the object types, although place-holders must be used.

target This identifies what the object represents. This identifies which objects correspond to the *target* parameter of the **Gg** (obsolete) and **Gx** operators. Some are deduced from their location in the file; others must be specifically identified. The objects that must be identified are:

1	all series, including legends
2	one sereis, including legends
4	one data bar/line/wedge
5	all data marks
6	one series' and its legends' marks
8	one data line segment's mark
9	one axis, including text, ticks, line
10	category axis' main line
15	all legend's text
20	all labels along category axis
22	entire "shadow" object

GS

The graph is finished: both the functional specification and the objects representing the graph have been written out. This is not unlike the U operator, which finishes a group.

11 Script Trailer

The syntax for the script trailer of an Adobe Illustrator document is as follows.

```
<document trailer> ::= %%Trailer
                       {<proc set termination>}*

<proc set termination> ::=
                       proc_set_name /terminate get exec
```

For each procedure set (resource) that was initialized in the script setup, the trailer invokes its terminate procedure in reverse order.

```
Adobe_Illustrator /terminate get exec
   Adobe_pattern /terminate get exec
   Adobe_customcolor /terminate get exec
   Adobe_cshow /terminate get exec
   Adobe_cmykcolor /terminate get exec
```

12 Creating Illustrator Documents

Other applications may create documents to be edited subsequently by Adobe Illustrator. Illustrator requires very few of the prolog, setup, and trailer components, however, these documents cannot be printed until they are opened and re-saved within Adobe Illustrator. Figure 4 shows a minimal document that is still acceptable by Adobe Illustrator.

Figure 4 . *A minimal document acceptable by Adobe Illustrator*

```
<minimal document> ::=
                        %!PS-Adobe-3.0
                        %%BoundingBox: <llx> <lly> <urx> <ury>
                        %%EndComments
                        %%EndProlog
                        %%BeginSetup
                        <font encoding>
                        <pattern defs>
                        %%EndSetup
                        {<object>}*
                        %%Trailer
                        %%EOF
```

Section section 13.2" describes the format for the document template information required by Adobe Illustrator on the Macintosh computer. When opening a document without this template information, Adobe Illustrator displays a dialog box that prompts the user for the location of the template for the file (if any). By supplying the resource information directly in your minimal documents, you can avoid this slight inconvenience.

13 Illustrator on the Macintosh

On the Macintosh computer, the resource fork of an Adobe Illustrator document contains several ancillary resources. The following sections describe these Macintosh-based resources.

13.1 PICT Resource

An Adobe Illustrator document may have a graphical screen representation provided so that a preview of the illustration may be manipulated on the screen by a page composition system. On the Macintosh, this representation is saved as a QuickDraw *PICT* picture resource within the resource fork of the document. The resource is given a resource type of *PICT* and a resource number of 256.

The picture's *picFrame* bounding box matches the bounding box of the illustration. This bounding box is specified in the **%%BoundingBox** comment in the prolog header. This is, the width and height of the *picFrame* are the same as the width and height, respectively, of the bounding box.

The picture resource is composed of two bitmap images: the image itself and its mask. If a particular bit is set in the mask, then the illustration has actually been painted the corresponding bit of the image. Otherwise, the corresponding bit has not been painted and should be considered transparent.

The mask is placed in the picture first, in the QuickDraw *srcBic* mode. It "punches a white hole" in just those areas that are painted. Then the image is placed in the QuickDraw *srcOr* mode, which fills in the punched areas, but leaves the other areas untouched.

13.2 TEMP Resource

This resource identifies the name of the document's template file, if it has one. The resource has three components (in order):

1. A 32-bit integer containing the directory identifier of the folder containing the template file. The integer is zero if the document has no template.

2. A Pascal string containing the name of the volume on which the template file resides. The string is empty if the document has no template.

3. A Pascal string containing the name of the template file itself. The string is empty if the document has no template.

The resource is given a type of *TEMP* and a resource number of 256.

13.3 PAGE Resource

This resource contains the *x* and *y* coordinates of the document's page origin as specified by the *Page* tool in Adobe Illustrator. The coordinates are in the default user coordinate system. In this system, the unit length along both axes is $^1\!/_{72}$ inch. The resource consists of two 32-bit fixed point numbers. The first specifies the *y* coordinate and the second specifies the *x* coordinate. The resource is given a type of *PAGE* and a resource number of 256

13.4 PREC Resource

This resource contains the standard 120 byte *Macintosh Printing Manager* print record. It describes the document's user-specified printing references as selected from the *Page Setup* and *Print* dialog boxes. The resource is given a type of *PREC* and a resource number of 256.

14 Save Options and Their Formats

Illustrator 3.x can save a file with or without a preview illustration and in several different ways. The last four methods write the same data fork to the file and vary in how and what they write to the resource fork.

- *No Preview omit EPSF header.* This option places no Preview in the file.
- *No Preview include EPSF header.* This option places no Preview in the file, but includes an EPSF header for the file.
- *B&W Macintosh PICT.* This option writes a Preview in the form of a black and white Macintosh PICT resource to the resource fork of the file.
- *Color Macintosh PICT.* This option writes a Preview in the form of a color Macintosh PICT resource to the resource fork of the file.
- *IBM PC (TIFF)*. This option writes a binary file with a pointer to TIFF data.

15 Adobe Illustrator for Windows, Version 4.x

Adobe Illustrator for Windows Version 4.x implements a feature generally called *Illustrator Format on the Clipboard* to facilitate cutting and pasting complex Bézier-based artwork and text effects between applications. The data for Illustrator Format on the Clipboard is actually a complete ASCII Adobe Illustrator file as defined in this document.

In Microsoft Windows, the ASCII clipboard format name used for Illustrator Format on the Clipboard is “ADOBE AI3.” Programs which support Illustrator Format on the Clipboard should use this name in Microsoft Windows.

Currently, Adobe Illustrator for Windows Version 4.x supports cut, copy, and paste for Illustrator Format on the Clipboard. Adobe Streamline 3.0 for Windows and Adobe TypeAlign 2.0 for Windows support cut and copy of Illustrator Format on the Clipboard but do not support paste.

15.1 Header Changes Under Windows

Adobe Illustrator for Windows Version 4.x uses a subset of the full Adobe Illustrator header structure. Figure 5 lists the subset of comments used in the Windows version header.

Figure 5 . *Header contents of Adobe Illustrator for Windows Version 4.x*

```
%%Creator:  Adobe Illustrator(TM) version
  %%For:  (username) (organization)
  %%Title:  (illustration title)
  %%CreationDate:  (date) (time)
  %%DocumentProcSets:  Adobe_Illustrator_version level revision
  %%DocumentSuppliedProcSets:  Adobe_Illustrator_version level
  revision
  %%Documentfonts:  font…
  %%+font…
  %%BoundingBox: llx lly urx ury
  %%TemplateBox: llx lly urx ury
```

The principal difference for comments that convey arbitrary text information is how strings of characters are delimited. In documents of some versions of *Adobe Illustrator*, the individual strings are valid PostScript language strings; in other versions, the last text item in a comment is terminated by a newline character.

Adobe Illustrator for Windows Version 4.x requires four additional comments to provide information that is otherwise stored in the Macintosh resource fork by the Macintosh version of Adobe Illustrator.

%%Template: {*filename*}
%%PageOrigin: *x y*
%%PrinterName: {*printer brand name*}
%%PrinterRect: *llx lly urx ury*

%%Template: {filename}

The **%%Template** comment specifies the full pathname of the template for the illustration.

%%PageOrigin: x y

The **%%PageOrigin** comment specifies the coordinates of the document's page origin (in points) as specified by the *Page* tool. If you omit this comment, Illustrator sets the page origin to the ruler origin. See section 13.3" for more information about the page origin.

%%PrinterName: {printer brand name}

The **%%PrinterName** comment specifies the brand name of the printer for which the file is being generated—for example "Apple LaserWriter."

%%PrinterRect: llx lly urx ury

The **%%PrinterRect** comment specifies the bounding box of the image area of the printer identified in the **%%PrinterName** comment. *The coordinate system used by this comment is 4th quadrant rather than 1st quadrant as used in all other structuring comments.* This means that 0 on the y axis is at the upper left (rather than the lower left). If you omit this comment, Illustrator sets the default for letter size, portrait orientation as on the Apple LaserWriter. The Macintosh version of Adobe Illustrator ignores this comment.

15.2 Controlling the Grid in Windows and NeXT Versions

A grid allows objects to “snap to” an array of horizontal and vertical locations on the artwork.The capability of controlling the grid is part of Adobe Illustrator for Windows Version 4.x and Adobe Illustrator NeXT Version 3.x. Adobe Illustrator’s **%AI3_Grid** comment appears immediately after the **%%EndResource** comment and immediately before the **%%EndProlog** comment.

%AI3_Grid.*version hzSpace vtSpace gridSnap R G B visibility*

The **%AI3_Grid** comment specifies the appearance of the grid and whether an object should snap to grid lines.

version	This is the version number of the grid specification. It is an integer. Note the decimal point. The current version number is 0.
hzSpace	This parameter is the horizontal spacing between grid lines in display pixels. The grid origin is at the page origin.
vtSpace	This is the vertical spacing between grid lines in display pixels.
gridSnap	This parameter controls the grid snap: enabled, horizontal, vertical, or both. The least significant bit specifies horizontal snapping; 0 means snap off, 1 means snap on. The next bit specifies vertical snap, and the next after that specifies whether the grid is enabled or disabled.

0	no snap specified; snapping disabled
1	horizontal snap specified, snapping disabled
2	vertical snap specified, snapping disabled
3	horizontal and vertical snap specified, snapping disabled
4	no snapping specified, snapping enabled
5	horizontal snap specified, snapping enabled
6	vertical snap specified, snapping enabled
7	horizontal and vertical snap specified, snapping enabled

R G B	Three parameters specifying the Red Green Blue color for the grid, in the range of 0 to 1.
visibility	This is an integer that encodes grid visibility and position. The least significant bit controls the grid position: 1 means draw grid in front, 0 means draw it in back of everything else. The next digit encodes grid visibility. This value is independent of the grid color.

0	invisible grid drawn in back
1	invisible grid drawn in front
2	visible grid drawn in back
3	visible grid drawn in front

16 List of Operators

A	locked	
b	close, fill and stroke path	
B	fill and stroke path	
c	curveto	
C	curveto	
d	setdash	
D	polarized fill style	
E	pattern	
f	close & fill path	(implicit **newpath**)
F	fill path	(implicit **newpath**)
g	fill setgray	
G	stroke setgray	
h	closepath	
H	closepath	
i	setflat	
I	path text	
j	**setlinejoin**	
J	**setlinecap**	
k	fill **setcmykcolor**	
K	stroke **setcmykcolor**	
l	**lineto**	
L	**lineto**	
m	**moveto**	
M	**setmiterlimit**	

n	path neither filled nor stroked	(implicit **newpath**)
N	close path neither filled nor stroked	(implicit **newpath**)
O	fill overprint	
p	fill pattern	
P	stroke pattern	
q	group (that contains clipping)	
Q	ungroup (from group that contains clipping	
R	stroke overprint	
s	stroke closed path	(implicit **newpath**)
S	stroke path	(implicit **newpath**)
u	group	
U	ungroup	
v	**curveto**	
V	**curveto**	
w	**setlinewidth**	
W	clip	
x	fill custom color	
X	stroke custom color	
y	**curveto**	
Y	**curveto**	
Z	text	
`	illustration	
~	illustration	
_	null	
@	pattern ink	
&	pattern path	

*	guide object
^	KanjiBitMapFont (Only Japanese version)
$	KanjiBitMapFont (Only Japanese version)
!	KanjiBitMapFont (Only Japanese version)
#	KanjiBitMapFont (Only Japanese version)
*u	compound path group
*U	compound path ungroup
*w	wraparound group
*W	wraparound ungroup

16.1 Text Operators

To	begin text object	type To – type = 0 -> point text 1 -> area text 2 -> path text
TO	end text object	– TO – also pops text matrix

Tp	begin text path	a b c d tx ty startPt Tp –
		startPt = start point value
		matrix = anchor matrix (concated onto CTM by TP)

TP	end text path	– TP –
Tm	set text matrix	a b c d tx ty Tm –
Td	translate text matrix	tx ty Td –
TM	pop text matrix	– TM –
TR	reset pattern matrix	a b c d tx ty TR –
Tr	set render mode	render Tr – sets the render mode: render = 0 -> fill text 1 -> stroke text 2 -> fill and stroke text

		3 -> invisible text 4 -> mask & fill text text 5 -> mask & stroke text 6 -> mask, fill & stroke 7 -> mask (only) text 8 -> filled text with stroked text following (patterned text only) 9 -> stroked text (preceded by render mode 8 text; patterned text only)
Te	end render	– Te –
Tf	set font name and size	fontname size Tf –
Ta	set alignment	alignment To – alignment= 0 -> left aligned 1 -> center aligned 2 -> right aligned 3 -> justified 4 -> justified including last line
Tl	set leading	leading paragraphLeading Tl –
Tt	set user tracking	userTracking Tt –
TW	set word spacing	minSpace optSpace maxSpace TW –
Tw	set computed word spacing	wordSpace Tw
TC	set character spacing	minSpace optSpace maxSpace TC –
Tc	set computed char spacing	charSpace Tc –
Ts	set super/subscripting (rise)	rise Ts –
Ti	set indentation	firstStartIndent otherStartIndent stopIndent Ti –
Tz	set horizontal scaling	scalePercent Tz –
TA	set pairwise kerning	autoKern TA – autoKern = 0 -> no pair kerning 1 -> automatic pair kerning
Tq	set hanging quotes	hangingQuotes Tq – hangingQuotes = 0 -> no hanging quotes 1 -> hanging quotes

TE	Set std platform encoding	(encoding pairs) TE –
TZ	Set encoding vector	(optional encoding pairs) newFontNameLiteral oldFontNameLit-eral direction fontScript TZ –
Tx	non-justified text	textString Tx –
Tj	justified text	textString Tj –
TX	overflow text	textString TX –
TS	special chars	textString justified TS –
Tk	kern	autoKern kernValue Tk – autoKern = 0 -> manual kern 1 -> auto kern kernValue = kern value in em/1000 space
TK	non-printing kern	autoKern kernValue TK –
T*	translate matrix to start of new line	– T* –
T-	print a discretionary hyphen	- T– –
T+	discretionary hyphen	– T+ –

17 Document Syntax Summary

The notation <…>+ means one or more instances of the bracketed item. The notation {<…>}* means zero or more instances.

<document> ::=	<prolog> <script>
<prolog> ::=	%!PS-Adobe-3.0 EPSF-3.0 <header comments> %%EndComments {<proc set>}* (not required, normally present) %%EndProlog
<script> ::=	<setup> {<object>}* {<page trailer>}(not required, normally present) <document trailer> %%EOF
<setup> ::=	%%BeginSetup {%%IncludeFont: font}* {<proc set init>}* (not required, normally present <font encoding> <pattern defs> %%EndSetup
<pattern defs> ::=	{<pattern>}*
<pattern> ::=	%AI3_BeginPattern: (*patternname*) <E> %AI3_EndPattern
<page trailer> ::=	%%PageTrailer gsave annotatepage grestore showpage
<document trailer> ::=	%%Trailer {<proc set termination>}* (not required, normally present)
<object> ::=	{<A>}(object locking) <path object>| <path mask>| <composite object>| <text object>| <placed art object>| <subscriber object>| <graph object>| <PostScript document>

```
<path object> ::=        <paint style>
                         <path geometry>
                         <path render>|<*> (<*> indicates
                         guide operator)

<path mask> ::=          <paint style>
                         <path geometry>
                         <h>|<H>
                         <W>
                         <path render>

<paint style> ::=        {<color>|<overprint>|<path attributes>}*

<path attributes> ::=    <d>|
                         <D>|
                         <i>|
                         <j>|
                         <J>|
                         <M>|
                         <W>|
                         %AI3_Note: <note>

<note< ::=               up to 254 characters of arbitrary text.

<color> ::=              <fill color>|<stroke color>

<fill color> ::=         <g>|(fill black ink only, or)
                         <k>|(fill process ink, or)
                         <x>|(fill custom ink, or)
                         <p>(fill pattern)

<stroke color> ::=       <G>|(stroke black ink only, or)
                         <K>|(stroke process ink, or)
                         <X>|(stroke custom ink, or)
                         <P>(stroke pattern)

<overprint> ::=          <O>|(fill overprint, or)
                         <R>(stroke overprint)

<path geometry) ::=      <m>
                         {<path operator>}*

<path operator> ::=      <l>|<L>|<c>|<C>|<v>|<V>|<y>|<Y>

<path render> ::=        <N>|(closepath; no fill no stroke)
                         <n>|(neither fill nor stroke)
                         <F>|(fill)
                         <f>|(closepath; fill)
                         <S>|(stroke)
                         <s>|(closepath; stroke)
                         <B>|(fill and stroke)
                         <b>(closepath; fill and stroke)
```

```
<composite object> ::=
                        <group object>|
                        <group with a mask>|
                        <compound path>|
                        <compound path mask>|
                        <wraparound group>

<group object> ::=      <u>
                        <object>+
                        <U>

<group with a mask> ::=
                        <q>
                        {<object>}*
                        {<masked object>}*
                        <Q>

<masked object> ::=     <mask>|<object>

<mask> ::=              <path mask>|<compound path mask>

<compound path> ::=     <*u>
                        <compound path element>+
                        <*U>

<compound path element> ::=
                        <path object>|<compound group>

<compound group> ::=
                        <u>
                        <compound path element>+
                        <U>

<compound path mask> ::=
                        <*u>
                        <compound path mask element>+
                        <*U>

<compound path mask element> ::=
                        <path mask>|<compound mask group>

<compound mask group> ::=
                        <compound mask bottom group>|
                        <compound mask non-bottom group>

<compound mask bottom group> ::=
                        {<A>}
                        <q>
                        <path mask>+
                        <Q>
```

```
<compound mask non-bottom group> ::=
                         {<A>}
                         <u>
                         <compound mask group>+
                         <U>

<wraparound group> ::=
                         <*w>
                         {<object>}*
                         {<wraparound objects>}*
                         <*W>

<wraparound objects> ::=
                         <text object>|<object>

<text object> ::=        <To>
                         <text at a point>|
                         <text area>|
                         <text along a path>
                         <TO>

<text at a point> ::=    <Tp>
                         <TP>
                         <text run>+

<text area> ::=          <text area element>+
                         {<overflow text>}

<text area element> ::=
                         <Tp>
                         <path object>
                         <TP>
                         <text run>+

<text along a path> ::=
                         <Tp>
                         <path object>
                         <TP>
                         <text run>+
                         {<overflow text>}*

<text run> ::=           {<text style>|
                         <paint style>|
                         <text positions>|
                         <Tk>}*
                         <text body>
```

```
<text style> ::=            <Tr>|(render mode)
                            <Tf>|(font & size)
                            <Ts>|(rise and fall)
                            <Tz>|(horizontal scaling)
                            <Tt>|(tracking)
                            <TA>|(automatic kerning)
                            <TC>|(intercharacter spacing)
                            <TW>|(interword spacing)
                            <Ti>|(indents)
                            <Ta>|(alignment)
                            <Tq>|(hanging quotations)
                            <Tl>(leading)

<text position> ::=         (printing only)
                            <Tc>|(computed intechar spacing)
                            <Tw>|(computed interword space)
                            <Tm>|(text matrix)
                            <Td>|(translate)
                            <T*>|(translate down)
                            <TR>(reset matrix; found only in
                            pattern prototypes)

<text body> ::=             <Tx>|<Tj>|<T+>|<T–>

<overflow text> ::=         {<text style>|<paint style>|<TK>}*
                            <TX>|<T+>

<font> ::=                  [
                            {<re-encoding pairs>}*
                            <Te>
                            {<re-encoding>}*

<re-encoding> ::=           %AI3_BeginEncoding
                            newFontName oldFontName
                            <TZ>
                            %AI3_EndEncoding <font type>

<font type> ::=             AdobeType|TrueType

<placed art object> ::=
                            <'>
                            <art reference>
                            <~>

<art reference> ::=         <file reference>|<file inline>

<file reference> ::=        %%IncludeFile: <filename>

<file inline> ::=           %%BeginDocument:<filename>
                            … included file contents
                            %%EndDocument

<filename> ::=              platform-specific path name of file
```

```
<subscriber object> ::=
                    %AI3_Subscriber:<subscriber ID>
                    <placed art object>

<subscriber ID> ::= resource number of SECT resource in file

<graph object> ::=  <Gs>
                    <graph functional spec>
                    {<graph customizations>}
                    <graph group object>
                    <GS>

<graph functional spec> ::=
                    <graph size and dialog values>
                    {<graph subscriptions>
                    <graph axis>
                    <graph axis>
                    <graph axis>
                    <graph table specs>

<graph size and dialog values> ::=
                    <Gb>
                    <Gy>
                    <Gd>

<graph axis> ::=    <Ga>
                    <GA>

<graph table specs> ::=
                    {<Gw>}*
                    <Gz>
                    <Gc>+
                    <GC>

<graph customizations> ::=
                    <Gt>
                    {<graph customization>}*
                    <GT>

<graph customization> ::=
                    {<graph customization operator>}*
                    <GX>|<Gg>
                    {<Gv>}
```

```
<graph customization operator> ::=
                    {<Gm>}
                    {<Gf>}
                    {<Gy>}
                    {<GD>}
                    {<Ge>}
                    {<G1>}(gee-one)
                    {<Gi>}
                    {<Gl>}(gee-ell)
                    {<Gp>}
                    {<Gx>}
                    {Gr>}
                    {<G+>}
                    {<Gg>}
                    {<A>}
                    {<paint style>}*
                    {<text style>}

<graph subscriptions> ::=
                    <Gj>

<graph group object> ::=
                    <u>
                    {<graph rendered object>}*
                    <U>

<graph rendered object> ::=
                    <object>|<graph group object>
                    {<Go>}

<PostScript document> ::=
                    %%BeginDocument: <comment>
                    (arbitrary PostScript code)
                    %%EndDocument

<comment> ::=       arbitrary text
```

Appendix: Changes Since Earlier Versions

Changes since May 4, 1991 version

- Updated entire document to include information on the Adobe Illustrator 3.x and 4.x file formats--DRAFT specification only.

- Reformatted in the new document format.

Changes since July 18, 1990 version

- The description for the operator **q** was changed from “except that the first object in the group specifies” to “except that some objects in the group specify.”

- The cover addresses were updated.

Changes since December 29, 1989 version

- The descriptions for the operators **o** and **a** in section 5.7 have been corrected. (They were reversed.)

Metafile Header

Creation Date 10/21/94
Mod Date 4/7/95

Type	**Parent Hierarchy** 3DMF
	Binary: 3DMF 0x33444D46 — **Ascii:** 3DMetafile
Size	16
Parent Objects	none

Data Format

```
Uns16         majorVersion
Uns16         minorVersion
MetafileFlags flags
FilePointer   tocLocation

• As of this release:

majorVersion = 0
minorVersion = 8
• The final release of the metafile will be:

majorVersion = 1
minorVersion = 0

• MetafileFlags bitfield is:

Binary       Text
0x00000000   Normal
0x00000001   Stream
0x00000002   Database
```

SubObjects

none

Description

• The **metafile header** is the first object to appear in any metafile.
• Metafile **versions** of 1.x are expected to maintain some degree of compatibility.
• **Flags** indicate to a general degree of how the file is structured or should be read.

A **database** file indicates that the metafile is a library, and all objects that are “shared” appear in the table of contents.
A **stream** file indicates that no references exist in the metafile, so that a parsing program may discard encountered data when it is through with it.

If the **toc location** (**Table of Contents** location) is NULL, the entire file must be parsed to find a **Table Of Contents**.

Example

```
3DMetafile (
  1 0 # version
  Normal
  toc>
)
...
toc: TableOfContents (
  ...
)
```

● Begin Group

Creation Date 2/24/95
Mod Date 3/1/95

Type

Parent Hierarchy 3DMF

Binary: bgng 0x62676E67

Ascii: BeginGroup

Size

sizes of contained objects + (8 * number of child objects)

Parent Objects

special

Data Format

SubObjects

Description

The begin group object is used similarly to the container object, except it is used as the starting delimiter for a group. This allows a naive parser to traverse a metafile without special casing the many types of groups that appear in the metafile spec. It also allows for a single mechanism that is used to declare a group.

Please note that all objects of type "group" MUST be contained in a begin group, to allow them to be identified as starting a group.

Example

```
BeginGroup ( DisplayGroup ( ) )
  Triangle ( 0 0 1 0 0 0 0 1 0 )
  Translate ( 1 2 3 )
  Sphere ( )
EndGroup ( )

BeginGroup (
  OrderedDisplayGroup ( )
  DisplayGroupState ( DoNotDraw )
)
  Triangle ( 0 0 1 0 0 0 0 1 0 )
  Translate ( 1 2 3 )
  Sphere ( )
EndGroup ( )

BeginGroup ( InfoGroup ( ) )
  CString ( "Copyright (c) 1995" )
EndGroup ( )
```

● Container

Creation Date 10/27/94
Mod Date 1/24/95

Type	**Parent Hierarchy** 3DMF	
	Binary: cntr 0x636E7472	**Ascii:** Container
Size	sizes of contained objects + (8 * number of child objects)	
Parent Objects	special	

Data Format

SubObjects

special

Description

• Used to bind objects together to form a single object.
• Container objects always contain other objects.
• The first object in the container is called the “root” object, and sets the scope of the remaining objects in the container, called “subobjects.”
• In general, the “root” object instantiates the object with its default values, and subobjects append information to the original “root” object.
• There is one exception to these encapsulation rules, which is “group” objects. Although “group” objects contain a list of other objects, they are delimited with another 3DMF object, the end group object.

Example

```
Container (
  Box ( )
  Container (
    AttributeSet ( )
    DiffuseColor ( 1 0 1 )
  )
)
```

End Group

Creation Date 10/21/94
Mod Date 3/15/95

Type

Parent Hierarchy	3DMF
Binary: endg 0x656E6467	**Ascii:** EndGroup

Size

0

Parent Objects

none

No Data

• Groups should be arranged into non-overlapping pairs of BeginGroup (group type/data) and an “EndGroup” object.

• All groups must be arranged into DAGs. (no cycles are permitted)

SubObjects

none

Description

This object is used as a delimiter for all **group** objects.

Example

```
# Empty group
BeginGroup ( OrderedDisplayGroup ( ) )
EndGroup ( )

# Group containing 1 object
BeginGroup ( DisplayGroup ( ) )
  Translate ( 1 2 3 )
  Sphere ( )
EndGroup ( )

# Inline group referenced elsewhere

REDColor:
BeginGroup (
  DisplayGroup ( )
  DisplayGroupState ( IsInline )
)
  Container (
    AttributeSet ( )
    DiffuseColor ( 1 0 0 )
  )
EndGroup ( )

BeginGroup ( DisplayGroup ( ) )
  Reference ( 1 ) # REDColor
  Cone () # Cone is RED
EndGroup ()
toc: TableOfContents (
  nextTOC> -1 2 0 12
  1
  1 REDColor>
)
```

REFERENCE

Creation Date 10/21/94
Mod Date 1/24/95

TYPE	**Parent Hierarchy** 3DMF	
	Binary: rfrn 0x7266726E	**Ascii:** Reference
SIZE	4	
PARENT OBJECTS	may be substituted for any Shared object	

DATA FORMAT

```
Uns32 refID

• if refID = 0, must contain subobjects

• if refID > 0, a TOC must exist in current
metafile that contains refID's resolution

• This refID is resolved in the current metafile
unless a Storage subobject is found in the
Reference
```

SUBOBJECTS

```
1 Storage object (optional)
```

DESCRIPTION

The **reference** object is used to instantiate an object multiple times in a metafile.

It may be substituted anywhere in the metafile for another "**Shared**" object. Only **shared** objects may be referenced.

References are resolved in the **Table Of Contents**. If a "Storage" object is specified as a subobject, it is assumed that the **reference** is external to the current metafile, and should be resolved in that external storage's table of contents.

EXAMPLE

```
Reference ( 23 ) # internal reference
...
toc: TableOfContents (
  nextTOC> 35 -1 0 12
  ...
  20 CarFrame>
  21 Axle>
  23 WheelOfCar>
  ...
)

Container ( # external reference
  Reference ( 23 )
  UnixPath ( "parts/car.eb" )
)
```

Table Of Contents

Creation Date 10/21/94
Mod Date 2/24/95

Type

Parent Hierarchy	3DMF
Binary: toc 0x746F6320	**Ascii:** TableOfContents

Size

28 + (tocEntrySize * nEntries)

Parent Objects

Data Format

```
FilePointer nextTOC
Uns32       refSeed
Int32       typeSeed
Uns32       tocEntryType
Uns32       tocEntrySize
Uns32       nEntries
TOCEntry    tocEntries

• refSeed > 0
• typeSeed < 0
• tocEntryType = 0 or 1
• tocEntrySize = 12 or 16, based upon tocEntryType
• the TOCEntry structure  is:
  - tocEntryType 0, tocEntrySize 12 is:

    Uns32       refID
    FilePointer objLocation

  - tocEntryType 1, tocEntrySize 16 is:

    Uns32       refID
    FilePointer objLocation
    ObjectType  objType
```

SubObjects

none

Description

The **table of contents** provides a means of resolving **references** within a file. The “nextTOC” file pointer points to the next **table of contents** in the file, or is NULL if no other **table of contents** exists.

The reference seed indicates the next available reference id available for **reference** objects. It is an unsigned positive number that is incremented with each addtional reference in a file. It is always one more than the maximum reference seed in a file.

The type seed indicates the next available type ID available for **type** objects. It is a negative number that is decremented with each additional **type** in a file. It is always one less than the minimum type seed in a file.

The **tocEntryType** and **tocEntrySize** are a set of paired values which indicate the size and type of information stored in a tocEntry.

The **tocEntries** are sorted by reference ID, in increasing order, to allow fast searching of the table of contents.

Example

```
3DMetafile (
  1 0
  Normal
  toc>
)
box23:
Mesh (
  45 # nVertices
  ...
)
Reference ( 1 )
Arrows:
BeginGroup ( DisplayGroup ( ) )
  Cone ( )
  Scale ( 0.2 0.1 0.2)
  Cylinder ( )
EndGroup ( )
Reference ( 2 )
Reference ( 4 )
...
Type ( -1 “Joe’s Garage:RepairHistory” )
...

-1 ( “Jim” “Fixed lug nut” 0.23 0.2 1.2 )

toc:
TableOfContents (
  nextTOC>
  5  # refSeed
  -2 # typeSeed
  0 12 # tocEntry Type/Size
  3 # nEntries
  1 box23>
  2 Arrows>
  4 Geom34>
)
```

Type

Creation Date 10/24/94
Mod Date 1/24/95

Type	**Parent Hierarchy** 3DMF	
	Binary: type 0x74797065	**Ascii:** Type
Size	4 + sizeof(String)	
Parent Objects		

Data Format

```
Int32    typeID
String   owner

• typeID < 0
• owner string
```

SubObjects

none

Description

A **type** definition is used to declare a custom data type. A **type** definition may appear anywhere in a file, however, the custom type must be encountered before the custom object of that type is encountered..

All custom types in the metafile are negative numbers, and the typeID field begins at -1 and is decremented for each additional type. Only 2147483648 (or 2^31) custom types are permitted in a single metafile.

The owner string is an ISO 9070 registered owner string. Owner strings are unique globally for each type of custom data.

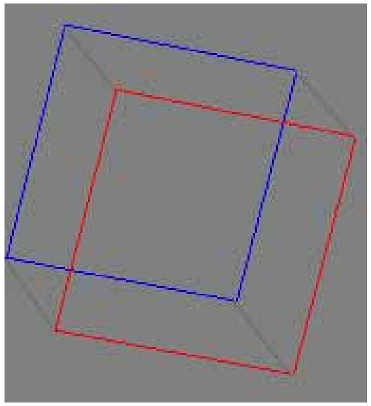
In the binary and text metafile, the typeID is used as the object type later in the file.

Example

```
Type (
  -1
  "Joe's Garage:BoltData"
)

...

-1 (
   -2.3 34 # Stress (kPA/area)
)
```

● FACE ATTRIBUTE SET LIST

Creation Date 10/21/94
Mod Date 3/15/95



TYPE

Parent Hierarchy	Data, AttributeSetList	
Binary: fasl	0x6661736C	**Ascii:** FaceAttributeSetList

SIZE

```
12 + nIndices * sizeof(Uns) + padding
```

PARENT OBJECTS

```
ALWAYS: Box, GeneralPolygon, Mesh, TriGrid
```

DATA FORMAT

```
Uns32         nObjects
PackingEnum   packing
Uns32         nIndices
Uns32         indices[nIndices]
```

• nObjects must match parent values

• PackingEnum is:

Binary	Text
0x00000000	Include
0x00000001	Exclude

• 0 ≤ indices < nObjects

SUBOBJECTS

many AttributeSet (order-dependent)

DESCRIPTION

The **face attribute set list** specifies a list of attributes to be attached to a set of faces determined by the parent's topology.

nObjects indicates the total number of objects being mapped to.

packing indicates how AttributeSet objects are mapped to indices. **Include** packing lists the face indices, in sequential order, of those faces to be assigned face attribute sets. **Exclude** packing lists the face indices, in sequential order, of those faces to NOT be assigned face attribute sets.

So, for example, supposing **nObjects** was 5, **Include** packing with a list of 3 indices after it means that there are 3 subobjects, each assigned to the indices in their order. **Exclude** packing with a list of 3 indices after it means there are 2 attribute sets subobjects, assigned to the indices NOT in the exclude list, in order.

The face attribute set list is padded to the nearest long word.

The values in **indices** always appear in increasing order.

If a packing value other than **Include** or **Exclude** is found, this object and its subobjects should be ignored.

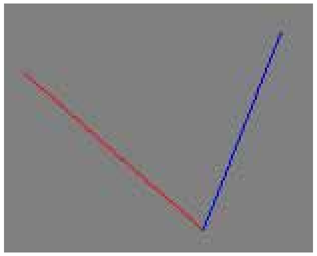
EXAMPLE

```
Container (
  Box ( )
  Container (
    FaceAttributeSetList (
      6 Include 2
      0 1
    )
    Container ( # assigned to 0
      AttributeSet ( )
      DiffuseColor ( 1 0 0 )
    )
    Container ( # assigned to 1
      AttributeSet ( )
      DiffuseColor ( 0 0 1 )
    )
  )
)

Container (
  Box ( )
  Container (
    FaceAttributeSetList (
      6 Exclude 2
      2 4
    )
    Container ( # assigned to 0
      AttributeSet ( )
      DiffuseColor ( 1 0 0 )
    )
    Container ( # assigned to 1
      AttributeSet ( )
      DiffuseColor ( 1 1 0 )
    )
    Container ( # assigned to 3
      AttributeSet ( )
      DiffuseColor ( 1 0 1 )
    )
    Container ( # assigned to 5
      AttributeSet ( )
      DiffuseColor ( 0 0 1 )
    )
  )
)
```

Geometry Attribute Set List

Creation Date 10/21/94
Mod Date 3/15/95

Type	**Parent Hierarchy**	Data, AttributeSetList
	Binary: gasl 0x6761736C	**Ascii:** GeometryAttributeSetList
Size	12 + nIndices * 4 + padding	
Parent Objects	ALWAYS: PolyLine	



Data Format

```
Uns32         nObjects
PackingEnum   packing
Uns32         nIndices
Uns32         indices[nIndices]
```

• nObjects must match parent values

• PackingEnum described in FaceAttributeSetList

SubObjects

many AttributeSet (order-dependent)

Description

The **geometry attribute set list** specifies a list of attributes to be attached to a set of geometric entities determined by the parent's topology.

Currently, only the **PolyLine** primitive uses this object. Each **attribute set** is mapped to a line segment in the **PolyLine**.

Packing for this object is identical to the other attribute set lists.

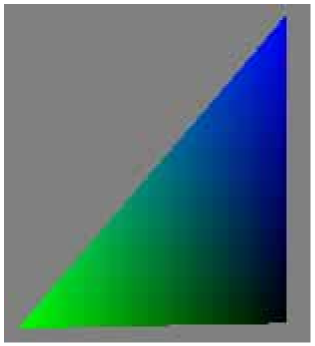
Example

```
Container (
  PolyLine (
   3
   10 2 3
   0 0 0
   2 8.5 3
  )
  Container (
    GeometryAttributeSetList (
      3 Exclude 1 1
    )
    Container ( # segment 0
      AttributeSet ( )
      DiffuseColor ( 1 0 0 )
    )
    Container ( # segment 2
      AttributeSet ( )
      DiffuseColor ( 0 0 1 )
    )
  )
)
```

● Vertex Attribute Set List

Creation Date 10/21/94
Mod Date 3/15/95

Type	**Parent Hierarchy** Data, AttributeSetList	
	Binary: vasl 0x7661736C	**Ascii:** VertexAttributeSetList
Size	12 + nIndices * sizeof(Uns) + padding	
Parent Objects	ALWAYS: GeneralPolygon, Line, Mesh, Polygon, PolyLine, Triangle, TriGrid	



Data Format

```
Uns32        nObjects
PackingEnum  packing
Uns32        nIndices
Uns32        indices[nIndices]
```

• nObjects must match parent values

• PackingEnum described in FaceAttributeSetList

SubObjects

many AttributeSet (order-dependent)

Description

The **vertex attribute set list** specifies a list of attributes to be attached to a set of vertices determined by the parent's topology.

Packing for this object is identical to the other attribute set lists.

Example

```
Container (
  Triangle (
    0 0 0
    0 2 0
    0 0 2
  )
  Container (
    VertexAttributeSetList (
      3 Exclude 0
    )
    Container ( # vertex 0
      AttributeSet ( )
      DiffuseColor ( 0 0 0 )
    )
    Container ( # vertex 0
      AttributeSet ( )
      DiffuseColor ( 0 0 1 )
    )
    Container ( # vertex 0
      AttributeSet ( )
      DiffuseColor ( 0 1 0 )
    )
  )
)
```

Camera Placement

Creation Date
Mod Date 1/24/95

Type

Parent Hierarchy Data, CameraData

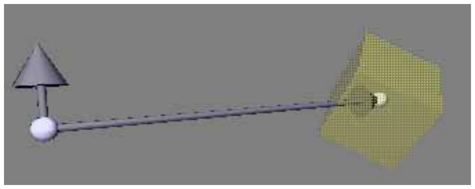
Binary: cmpl 0x636D706C	**Ascii:** CameraPlacement

Size

36

Parent Objects

ALWAYS: Camera objects: ViewAngleAspectCamera, ViewPlaneCamera, OrthographicCamera



Data Format

```
Point3D   location
Point3D   pointOfInterest
Vector3D  upVector
```

• upVector ⊥ (pointOfInterest - location)

• |upVector| = 1.0

• Default Values:

```
    0 0 1 # location
    0 0 0 # pointOfInterest
    0 1 0 # upVector
```

SubObjects

none

Description

The **camera placement** specifies the location and orientation of the camera in space, by a camera **location**, a **point of interest**, and an **up vector**. This placement locates and orients the camera, and defines a space in which the rest of the parameters are interpreted.

If the **up vector** is not of unit length upon reading, it should be normalized by the reading program.

The **camera placement** is affected by the current transformation state in a hierarchy. The **location** and **point of interest** are multiplied by the current transformation directly, and the **up vector** is multiplied by the current transformation minus any translation component of the transform, and unitized.

The **camera vector** is defined as:
camera vector = (**pointOfInterest** - **location**)

Example

```
Container (
  OrthographicCamera (
    -1 -1 1 1
  )
  CameraPlacement (
     10 0 0 # located along X axis
      0 0 0 # point of interest is origin
      0 1 0 # Y is up
  )
)
```

Camera Range

Creation Date	10/21/94
Mod Date	3/15/95

Type

Parent Hierarchy	Data, CameraData
Binary: cmrg 0x636D7267	**Ascii:** CameraRange

Size

8

Parent Objects

ALWAYS: Camera objects: ViewAngleAspectCamera, ViewPlaneCamera, OrthographicCamera

Data Format

```
Float32   hither
Float32   yon

• 0 < hither ≤ yon

• default is:

  hither  ε
  yon     ∞
```

SubObjects

none

Description

The **camera range** affects the clipping of the viewing frustum.

This is used to bound the range of the set of objects of interest.

Hither is the frontmost clipping plane (sometimes referred to as “near”), **yon** is the backmost clipping plane (sometimes referred to as “far”).

Each of these distances is measured along the **camera vector**, described in the **Camera Placement** object.

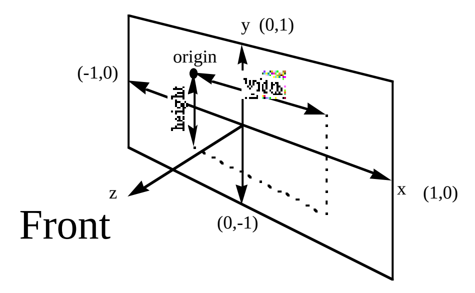
Example

```
Container (
  OrthographicCamera (
    -1 -1 1 1
  )
  CameraRange (
     0.1 2 # hither, yon
  )
)
```

CAMERA VIEWPORT

Creation Date 10/21/94
Mod Date 3/15/95

TYPE	**Parent Hierarchy** Data, CameraData	
	Binary: cmvp 0x636D7670	**Ascii:** CameraViewPort
SIZE	16	
PARENT OBJECTS	ALWAYS: any Camera object: ViewAngleAspectCamera, ViewPlaneCamera, OrthographicCamera	



DATA FORMAT

```
Point2D   origin
Float32   width
Float32   height
```

- -1 ≤ origin.x ≤ 1
- -1 ≤ origin.y ≤ 1
- 0 < width ≤ 2
- 0 < height ≤ 2
- Default is:

```
  -1 1  # origin
  2      # width
  2      # height
```

SUBOBJECTS

none

DESCRIPTION

The **camera viewport** specifies a rectangular region of the viewing frustum to which the image is clipped. Effectively the **view port** may be used to zoom in on a particular feature of an image.

The view port uses the cartesian coordinate system, with Y towards the top of the screen, X to the right, and Z coming towards the viewer, as shown in the diagram.

EXAMPLE

```
Container (
  OrthographicCamera (
    -1 -1 1 1
  )
  CameraViewPort ( # zoom to 200%
    -0.5 0.5 1 1
  )
)
```

Bottom Cap Attribute Set

Creation Date 10/21/94
Mod Date 4/7/95

Type

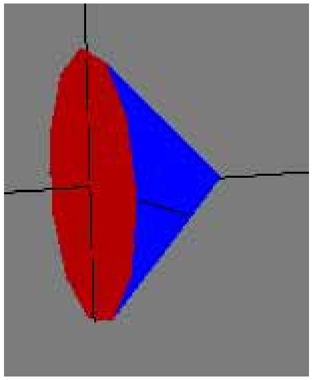
Parent Hierarchy	Data, CapData
Binary: bcas 0x62636173	**Ascii:** BottomCapAttributeSet

Size

0

Parent Objects

ALWAYS: Cone, Cylinder



No Data

SubObjects

1 AttributeSet (optional)

Description

This object simply allows the attributes associated with the bottom cap of a **Cone** or **Cylinder** to be encapsulated.

Presence of a **bottom cap attribute set** does not neccessarily mean the bottom cap is drawn.

The **Caps** object determines whether the **Cone** and **Cylinder** caps are drawn or not.

Example

```
3DMetafile ( 1 0 Normal toc> )
Container (
  Cone ( )
  Caps ( Bottom )
  Container (
    BottomCapAttributeSet ( )
    capColor: Container (
      AttributeSet ( )
      DiffuseColor ( 1 0 0 )
    )
  )
)
Container (
  Cone ( )
  Caps ( Bottom )
  Container (
    BottomCapAttributeSet ( )
    Reference (1)
  )
)
...
toc: TableOfContents (
   ...
   1 capColor>
)
```

Caps

Creation Date 10/21/94
Mod Date 1/24/95

Type

Parent Hierarchy	Data, CapData
Binary: caps 0x63617073	**Ascii:** Caps

Size

4

Parent Objects

ALWAYS: Cone, Cylinder

Data Format

```
CapsFlags caps

• CapsFlags is defined as:
```

Binary	Text
0x00000000	None
0x00000001	Bottom
0x00000002	Top

```
• Default is:
  None
```

SubObjects

none

Description

In the binary file, the upper 28 bits of the **caps** bitfield should be ignored. In the text file, unknown bitfield strings should be skipped. The default **caps** value is **0**, or **None**.

The **Top** cap bit (label) is ignored in the **Cone**.

Example

```
Container (
  Cylinder ( )
  Caps ( Bottom | Top )
)

Container ( # Cone with a blue bottom
  Cone ( )
  Caps ( Bottom )
  Container (
    BottomCapAttributeSet ( )
    Container (
      AttributeSet ( )
      DiffuseColor ( 0 0 1 )
    )
  )
)
```

Face Cap Attribute Set

Creation Date 10/21/94
Mod Date 1/24/95

Type

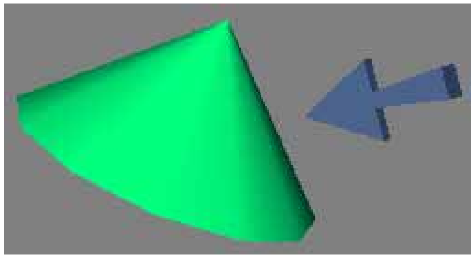
Parent Hierarchy	Data, CapData	
Binary: fcas	0x66636173	**Ascii:** FaceCapAttributeSet

Size

0

Parent Objects

ALWAYS: Cone, Cylinder



No Data

SubObjects

1 AttributeSet (optional)

Description

Attaches a set of attributes to the face “cap” of the **cone** and **cylinder** primitives. For the cone, it’s indicated in the diagram.

Example

```
Container (
  Cone ( )
  Caps ( Bottom )
  Container (
    FaceCapAttributeSet ( )
    Container (
      AttributeSet ( )
      DiffuseColor ( 0.2 0.9 0.4 )
    )
  )
)
```

Top Cap Attribute Set

Creation Date 10/21/94
Mod Date 4/4/95

Type

Parent Hierarchy Data, CapData

Binary: tcas 0x74636173

Ascii: TopCapAttributeSet

Size

0

Parent Objects

ALWAYS: Cylinder

No Data

SubObjects

1 AttributeSet (optional)

Description

Attaches a set of attributes to the top “cap” of the **cylinder** primitive.

Presence of a **top cap attribute set** does not neccessarily mean the top cap is drawn.

The **Caps** object determines whether the **Cylinder** caps are drawn or not.

Example

```
Container (
  Cylinder ( )
  Caps ( Top )
  Container (
    TopCapAttributeSet ( )
    Container (
      AttributeSet ( )
      DiffuseColor ( 0.2 0.9 0.4 )
    )
  )
)
```

Display Group State

Creation Date 10/21/94
Mod Date 4/7/95

Type

Parent Hierarchy	Data
Binary: dgst 0x64677374	**Ascii:** DisplayGroupState

Size

4

Parent Objects

ALWAYS: DisplayGroup, OrderedDisplayGroup

Data Format

```
DisplayGroupStateFlags  traversalFlags

• DisplayGroupStateFlags is:

Binary     Text
0x00000000 None
0x00000001 Inline
0x00000002 DoNotDraw
0x00000004 NoBoundingBox
0x00000008 NoBoundingSphere
0x00000010 DoNotPick

• default is:

Binary     Text
0x00000000 None
```

SubObjects

none

Description

This piece of data is a subobject only to objects of type **display group**. It affects how a **display group** is traversed. These flags allow any **display group** to have the following characteristics:

• To have "invisible" objects in a scene which may act as user interface items, or may aid in bounding complex geometries
• To have non-user interface items which may serve only as decoration and should not be picked.
• To have a group of shaders/attributes which affects the state as an inline group so it may be instantiated and inherited in many parts of a hierarchy.

Example

```
# to pick a chess piece by a box around it

BeginGroup ( DisplayGroup ( ) )
  PickIDStyle ( 1 )
  BeginGroup (
    DisplayGroup ( )
    DisplayGroupState ( DoNotDraw )
  )
    Scale ( 2 4 2 )
    Box ( )
  EndGroup ( )

  Container (
    DisplayGroup ( )
    DisplayGroupState ( DoNotPick )
  )
  Mesh ( # chess piece
    56 # nVertices
    0.2 0.3 0.5
    ...
  )
  EndGroup ( )
EndGroup ( )
```

General Polygon Hint

Creation Date 3/15/95
Mod Date 3/15/95

Type

Parent Hierarchy Data

Binary: gplh 0x67706C68

Ascii: GeneralPolygonHint

Size

4

Parent Objects

ALWAYS: GeneralPolygon

Data Format

```
GeneralPolygonHintEnum   shapeHint

• GeneralPolygonHintEnum is:

Binary     Text
0x00000000 Complex
0x00000001 Concave
0x00000002 Convex

• default is:
    Complex
```

SubObjects

none

Description

The **GeneralPolygonHint** gives a reading application some hint of what shape a general polygon is.

A “Complex” general polygon may contain intersecting, concave, or convex polygons.

A “Concave” general polygon contains no intersecting polygons, but contains 1 or more concave polygons.

A “Convex” general polygon indicates that all contained polygons are convex and non-intersecting.

Example

```
Container (
  GeneralPolygon (
     1
     3
     0 2 3
     0 2 1
     2 0 0
  )
  GeneralPolygonHint ( Convex )
)
```

● LIGHT DATA

Creation Date 10/21/94
Mod Date 10/27/94

TYPE

Parent Hierarchy	Data	
Binary: lght	0x6C676874	**Ascii:** LightData

SIZE

20

PARENT OBJECTS

ALWAYS: any Light: SpotLight, AmbientLight, PointLight, DirectionalLight

DATA FORMAT

```
Boolean   isOn
Float32   intensity
ColorRGB  color

• 0 ≤ intensity ≤ 1

• Default is:
   True  # isOn
   1.0   # intensity
   1 1 1 # color
```

DESCRIPTION

The **light data** object affects information about a light that is common among all lights.

A **light** may be on or off, may vary in intensity, or may have different colors.

EXAMPLE

```
Container (
  AmbientLight ( )
  LightData (
    True
    0.4
    1 0 0
  )
)
```

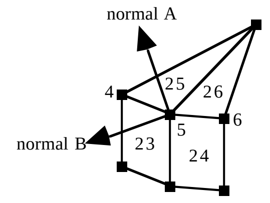
SUBOBJECTS

none

Mesh Corners

Creation Date 10/21/94
Mod Date 3/15/95

Type	**Parent Hierarchy** Data	
	Binary: crnr 0x63726E72	**Ascii:** MeshCorners
Size	4 + sizeof(corners[0..nCorners-1]) sizeof(MeshCorner) = 8 + nFaces * 4	
Parent Objects	ALWAYS: Mesh	



Data Format

```
Uns32       nCorners
MeshCorner  corners[nCorners]

• 0 < nCorners
• where MeshCorner is:

Uns32    vertexIndex
Uns32    nFaces
Uns32    faces[nFaces]

• 0 < nFaces
```

SubObjects

nCorners **AttributeSet**s (order-dependent)

Description

Mesh Corners allow you to attach **AttributeSet**s to a mesh vertex, to allow for attributes to be associated with a particular face-vertex pair. This may be used to allow sharp corners in an object (diagram above), to set different shading parameters for adjacent faces, etc.

Mesh corners supplies a vertex index, a list of face indices, and a **vertex attribute set** for each corner.

The **mesh corners** object most often appears inside a **container**, and always has **AttributeSet** subobjects. The first corner in the **mesh corners** data is mapped to the first **attribute set** subobject, the second corner to the second **attribute set**, etc.

Example

```
Container (
  Mesh (
    ...
  )
  Container (
    MeshCorners (
      2 # numCorners

      # Corner 0
      5 # vertexIndex
      2 # faces
      25 26 # face indices

      # Corner 1
      5 # vertexIndex
      2 # faces
      23 24 # face indices
    )
    Container (
      AttributeSet ( )
      Normal ( -0.2 0.8 0.3 )
    )
    Container (
      AttributeSet ( )
      Normal ( -0.7 -0.1 0.4 )
    )
  )
)
```

MESH EDGES

Creation Date 3/1/95
Mod Date 3/15/95

TYPE

Parent Hierarchy Data

Binary: edge 0x65646765

Ascii: MeshEdges

SIZE

```
4 + sizeof(corners[0..nCorners-1])

sizeof(MeshEdges) = 2 * sizeof(Uns)
```

PARENT OBJECTS

ALWAYS: Mesh

DATA FORMAT

```
Uns32        nEdges
MeshEdge     edges[nEdges]

• 0 < nEdges
• where MeshEdge is:

Uns32  vertexIndex1
Uns32  vertexIndex2
```

SUBOBJECTS

nCorners **AttributeSet**s (order-dependent)

DESCRIPTION

Mesh Edges allow you to attach **AttributeSet**s to a mesh edge.

You may attach mesh edges to any edge in the mesh that corresponds to a face edge. To specify and edge that should have an attribute set attached to it, include it as the nth edge the list of edges, and specify the attribute set as the nth attribute set subobject.

EXAMPLE

```
Container (
  Mesh (
    ...
  )
  Container (
    MeshEdges (
      2 # numEdges
      0 1 # 1st edge vertexIndices
      1 2 # 2nd edge vertexIndices
    )
    Container ( /* 1st edge attribute set */
      AttributeSet ( )
      DiffuseColor ( 0.2 0.8 0.3 )
    )
    Container ( /* 2nd edge attribute set */
      AttributeSet ( )
      DiffuseColor ( 0.8 0.2 0.3 )
    )
  )
)
```

● *NURB CURVE 2D*

Creation Date 10/21/94
Mod Date 1/24/95

TYPE	**Parent Hierarchy** Data	
	Binary: nb2c 0x6E623263	**Ascii:** NURBCurve2D
SIZE	8 + 12 * nPoints + 4 * (order + nPoints)	
PARENT OBJECTS	ALWAYS: TrimCurves	

DATA FORMAT

```
Uns32               order
Uns32               nPoints
RationalPoint3D     points[nPoints]
Float32             knots[order + nPoints]
```

- 2 ≤ order
- 2 ≤ nPoints
- 0 < points[...].w (weights of points)

SUBOBJECTS

none

DESCRIPTION

The **NURB Curve 2D** is a subobject of the **TrimCurves** object, and supplies a 2 dimensional curve to trim **NURB Patches**.

EXAMPLE

Shader Data

Creation Date 10/21/94
Mod Date 3/22/95

Type

Parent Hierarchy	Data
Binary: shdr 0x73686472	**Ascii:** ShaderData

Size

8

Parent Objects

ALWAYS: any Shader

Data Format

```
ShaderUVBoundaryEnum   uBounds
ShaderUVBoundaryEnum   vBounds
```

• ShaderUVBoundaryEnum is:

Binary	Text
0x00000000	Wrap
0x00000001	Clamp

• default is:
    Wrap Wrap

Description

The **shader data** initializes boundary wrapping conditions for a **shader**.

Example

```
Container (
  CustomShader ( ... )
  ShaderData ( Wrap Clamp )
)
```

SubObjects

none

Shader Transform

Creation Date 10/21/94
Mod Date 4/7/95

TYPE	**Parent Hierarchy** Data	
	Binary: sdxf 0x73647866	**Ascii:** ShaderTransform
SIZE	64	
PARENT OBJECTS	ALWAYS: any Shader	

Data Format

```
Matrix4x4   shaderTransform
```

SubObjects

none

Description

This transforms a shaded object into another world space coordinate system. It does not affect how the object is drawn, or the current state of the hierarchy.

Example

```
Container (
  3DMarbleShader ( )
  ShaderTransform (
    1 0 0 0
    0 1 0 0
    0 0 1 0
    2 3 4 1
  )
)
...
Type ( -3 “Apple:ATG:3DMarbleShader” )
Container (
  -3 ( 2.3 1.0 -10 )
  ShaderTransform (
    1 0 0 0
    0 1 0 0
    0 0 1 0
    2 3 4 1
  )
)
```

Shader UV Transform

Creation Date 10/21/94
Mod Date 3/15/95

Type

Parent Hierarchy	Data
Binary: sduv 0x73647576	**Ascii:** ShaderUVTransform

Size

36

Parent Objects

ALWAYS: any Shader

Data Format

```
Matrix3x3   matrix
```

SubObjects

none

Description

The **Shader UV transform** allows the uv’s on a geometric object to be transformed before shading occurs.

This allows you to rotate a texture map, for example.

Example

```
Container (
  TextureShader ( )
  ShaderUVTransform (
    1 0 0
    0 1 0
    0.2 0.3 1
  )
  PixmapTexture (
    ...
  )
)
```

● *Trim Curves*

Creation Date 10/21/94
Mod Date 4/7/95

Type	**Parent Hierarchy** Data	
	Binary: trml 0x74726D63	**Ascii:** TrimLoop
Size	0	
Parent Objects	ALWAYS: NURBPatch	

No Data

SubObjects

many NURBCurve2D (order-dependent)

Description

The **Trim Loop** subobject allows users to attach trimming loops to a **NURB Patch**. The **Trim Loop** object contains no data, and serves only as an encapsulation of various 2-dimensional curves used for trimming.

The Trim loop object contains a sequence of 2 dimensional curves which are “concatenated” together to form a loop. The subobjects are order-dependent. Each trim loop subobject should contain loops that are geometrically continuous, meaning the first trim curve’s end point ends at the next trim curve’s starting point.

In the metafile version 1.0, the only 2-dimensional curve allowed is a **NURBCurve2D**.

In future releases of the metafile, we expect to add additional types of 2d trim curves for trimming NURBS.

Example

```
Container (
 NURBPatch (
  4 4 4 4 # u,v order, num M,N points
  -2 2 0 1   -1 2 0 1   1 2 0 1   2 2 0 1
  -2 2 0 1   -1 2 0 1   1 0 5 1   2 2 0 1
  -2 -2 0 1  -1 -2 0 1  1 -2 0 1  2 -2 0 1
  -2 -2 0 1  -1 -2 0 1  1 -2 0 1  2 -2 0 1
   0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 # knots
 )
 Container (
  TrimLoop ( )
  NURBCurve2D (
    ...
  )
  NURBCurve2D (
    ...
  )
 )
)
```

Image Clear Color

Creation Date 1/20/95
Mod Date 1/24/95

Type

Parent Hierarchy	Data, ViewHintsData
Binary: imcc 0x696D6363	**Ascii:** ImageClearColor

Size

12

Parent Objects

ALWAYS: ViewHints

Data Format

```
ColorRGB   clearColor
```

SubObjects

none

Description

This specifies the preferred rgb color with should be used to clear the drawing area’s background.

Example

```
3DMetafile ( 1 0 Normal toc> )
Container (
  ViewHints ( )
  ImageClearColor ( 1 1 1 ) # white
)
Box ( )
```

IMAGE DIMENSIONS

Creation Date 1/20/95
Mod Date 1/24/95

TYPE	**Parent Hierarchy** Data, ViewHintsData	
	Binary: imdm 0x696646D	**Ascii:** ImageDimensions
SIZE	8	
PARENT OBJECTS	ALWAYS: ViewHints	

DATA FORMAT

```
Uns32   width
Uns32   height
```

- 0 < width
- 0 < height

SUBOBJECTS

none

DESCRIPTION

The **image dimensions** specifies the preferred image width and height in bits. It is a subobject of the **view hints**, which aids an application in determining how to display an image.

EXAMPLE

```
3DMetafile ( 1 0 Normal toc> )
Container (
  ViewHints ( )
  ImageDimensions ( 32 32 )
  ImageClearColor ( 1 1 1 )
)
Rotate ( X 0.75 )
Rotate ( Y 0.75 )
Container (
  AttributeSet ( )
  DiffuseColor ( 1 0 0 )
)
Box ( )
```

Image Mask

Creation Date 1/20/95
Mod Date 3/23/95

Type	**Parent Hierarchy** Data, ViewHintsData
	Binary: immk 0x696D6D6D — **Ascii:** ImageMask
Size	12 + (rowBytes * height) + padding
Parent Objects	ALWAYS: ViewHints

Data Format

```
Uns32       width
Uns32       height
Uns32       rowBytes
EndianEnum  bitOrder
RawData     image[rowBytes * height]
```

• width, height in bits
• 0 < width
• 0 < height
• ((width >> 3) + ((width & 0x7) ? 1 : 0)) ≤ rowBytes
• EndianEnum is:

Binary	Text
0x00000000	BigEndian
0x00000001	LittleEndian

SubObjects

none

Description

The **image mask** is a bitmap that specifies how an image's rendered pixels should be clipped. The origin of the bitmap (the upper-left) is aligned with the origin (upper left) of the drawing area. Generally, the **image mask** and the **image dimensions** are used simultaneously to specify an image which is partially clipped.

The example to the right specifies a mask to clip a 32x32 image. The application using this data uses this clip mask to only render to a clipped portion of a custom document icon – in this case, the bitmap will only draw inside of a "document" icon, providing a small preview in the Finder with a black document icon. The image mask to the right was used to render the example above.

Example

```
3DMetafile ( 1 0 Normal toc> )
Container (
  ViewHints ( )
  ImageDimensions ( 32 32 )
  ImageClearColor ( 1 1 1 )
  ImageMask (
    32 32 # width, height
    4 # rowBytes
    BigEndian # bitOrder
    0x000000000FFFF8000FFFF8000FFFF800
    0x0FFFF8000FFFF8000FFFF8000FFFFFE0
    0x0FFFFFE00FFFFFE00FFFFFE00FFFFFE0
    0x0FFFFFE00FFFFFE00FFFFFE00FFFFFE0
    0x0FFFFFE00FFFFFE00FFFFFE00FFFFFE0
    0x0FFFFFE00FFFFFE00FFFFFE00FFFFFE0
    0x0C61FFE00F24FFE00E64FFE00F24FFE0
    0x0F24FFE00C61FFE00FFFFFE000000000
  )
)
Rotate ( X 0.25 )
Rotate ( Y 0.23 )
Container (
  Torus ( 0 0.7 0 0 0 1 1 0 0 0 0 0 0.7 )
  Container (
    AttributeSet ( )
    DiffuseColor ( 0.2 0.9 0.9 )
  )
)
```

Ambient Coefficient

Creation Date 10/21/94
Mod Date 1/18/95

Type

Parent Hierarchy	Element, Attribute
Binary: camb 0x63616D62	**Ascii:** AmbientCoefficient

Size

4

Parent Objects

ALWAYS: AttributeSet

Data Format

`Float32 ambientCoefficent`

• 0 ≤ ambientCoefficient ≤ 1.0

SubObjects

none

Description

The **ambient coefficient** describes the intensity of the ambient light that is reflected by a surface.

Example

```
Container (
  AttributeSet ( )
  AmbientCoefficient ( 0.7 )
)
```

Diffuse Color

Creation Date 10/21/94
Mod Date 1/18/95

Type	**Parent Hierarchy** Element, Attribute
	Binary: kdif 0x6B646966 — **Ascii:** DiffuseColor
Size	12
Parent Objects	ALWAYS: AttributeSet

Data Format

```
ColorRGB   diffuseColor
```

Description

The **diffuse color** indicates the amount of diffuse light reflected by a surface.

Example

```
Container (
  AttributeSet ( )
  DiffuseColor ( 1 0 0 ) # red
)
```

SubObjects

none

Highlight State

Creation Date 10/21/94
Mod Date 3/15/95

Type

Parent Hierarchy	Element, Attribute
Binary: hlst 0x686C7374	**Ascii:** HighlightState

Size

4

Parent Objects

ALWAYS: AttributeSet

Data Format

```
Boolean  highlighted
```

SubObjects

none

Description

The **highlight state** attribute, when **true**, indicates that the current attribute state is overridden with the current **highlight style**'s **attribute set**. The **highlight state** attribute allows various portions of a **geometry** object to be highlighted for user interface, etc. while retaining the integrity of a **geometry**'s **attribute set**.

Example

```
Container (
  HighlightStyle ( )
  Container (
    AttributeSet ( )
    DiffuseColor ( 1 0 0 ) # RED
  )
)
...
Container (
  Polygon (
    3
    0 1 2
    0 0 0
    0 -1 2
  )
  Container (
    AttributeSet ( )
    DiffuseColor ( 0 1 2 )
    HighlightState ( True )
    # polygon is drawn RED
  )
)
```

Normal

Creation Date 10/21/94
Mod Date 1/18/95

Type	**Parent Hierarchy** Element, Attribute	
	Binary: nrml 0x6E726D6C	**Ascii:** Normal
Size	12	
Parent Objects	ALWAYS: AttributeSet	

Data Format

```
Vector3D  normal
```

• |normal| = 1

SubObjects

none

Description

If **normal** is not of unit length upon reading, it should be normalized. (npi)

The **normal** indicates the surface normal at a vertex.

Example

```
Container (
  Polygon (
     5
     0.23423 0.56434 0.2312
     ...
  )
  Container (
    VertexAttributeSetList ( 5 Exclude 0 )
    Container (
      AttributeSet ( )
      Normal ( 0.8 -0.1 -0.1 )
    )
  )
)
```

● *Shading UV*

Creation Date 10/21/94
Mod Date 3/15/95

◁ ▷

Type

Parent Hierarchy	`Element, Attribute`
Binary: `shuv` `0x73687576`	**Ascii:** `ShadingUV`

Size

8

Parent Objects

ALWAYS: AttributeSet

Data Format

```
Param2D  shadingUV

• Any UV parametrization is allowed, however,
shading generally occurs with the following values.

• 0 ≤ shadingUV.u ≤ 1
• 0 ≤ shadingUV.v ≤ 1
```

SubObjects

none

Description

The **shading UV** indicates an alternate UV to the **Surface UV** for shading purposes.

Shading UV's are generally used by **shaders** that affect appearance information, such as texture maps, which alter the color on a geometric surface.

Surface UV's are generally used for trimming.

Example

```
Container (
  AttributeSet ( )
  ShadingUV ( 0 0 )
)
```

Specular Color

Creation Date 10/21/94
Mod Date 3/15/95

Type	**Parent Hierarchy** Element, Attribute	
	Binary: kspc 0x6B737063	**Ascii:** SpecularColor
Size	12	
Parent Objects	ALWAYS: AttributeSet	

Data Format

```
ColorRGB   specularColor
```

SubObjects

none

Description

The **specular color** indicates the color of specular highlights on a surface.

Example

```
Container (
  AttributeSet ( )
  DiffuseColor ( 0.1 0.1 0.1 ) # near-black
  SpecularColor ( 1 1 1 ) # white
)
Sphere (
  0 0 0
  0 1 0
  1 0 0
  0 0 1
)
```

Specular Control

Creation Date 10/21/94
Mod Date 3/15/95

Type

Parent Hierarchy	Element, Attribute
Binary: cspc 0x63737063	**Ascii:** SpecularControl

Size

4

Parent Objects

ALWAYS: AttributeSet

Data Format

```
Float32  specularControl

• 0 ≤ specularControl
```

SubObjects

none

Description

The **specular control** attribute indicates the power to which the specular component of lighting computations is raised.

Example

```
Container (
  AttributeSet ( )
  DiffuseColor ( 0.5 0.5 0.5 ) # near-black
  SpecularColor ( 0.5 ) # white highlights
  SpecularControl ( 1 ) # larger highlight area
)
Sphere ( )
```

SURFACE TANGENT

Creation Date 10/21/94
Mod Date 3/15/95

TYPE	**Parent Hierarchy** Element, Attribute
	Binary: srtn 0x7372746E — **Ascii:** SurfaceTangent
SIZE	24
PARENT OBJECTS	ALWAYS: AttributeSet

DATA FORMAT

```
Vector3D  paramU
Vector3D  paramV
```

SUBOBJECTS

none

DESCRIPTION

The **surface tangent** attribute indicates the direction of changing U and V on a surface.

EXAMPLE

```
Container (
  Mesh (
    ...
  )
  Container (
    VertexAttributeSetList (
      ...
    )
    Container (
      AttributeSet ( )
      SurfaceUV ( 0.1 0.293 )
      SurfaceTangent (
        1 0 0
        0 1 0
      )
    )
  )
)
```

Surface UV

Creation Date 10/21/94
Mod Date 3/15/95

Type

Parent Hierarchy	Element, Attribute
Binary: sruv 0x73727576	**Ascii:** SurfaceUV

Size

8

Parent Objects

ALWAYS: AttributeSet

Data Format

```
Param2D  surfaceUV
```

• Any UV parametrization is allowed, however, shading generally occurs with the following values.

• 0 ≤ surfaceUV ≤ 1
• 0 ≤ surfaceUV ≤ 1

SubObjects

none

Description

The **surface UV** indicates an alternate UV to the **shading UV** for shading purposes.

Surface UV's are generally used for trim shaders.

Shading UV's are generally used by shaders that affect appearance information, such as texture maps, which alter the color on a geometric surface.

Example

```
Container (
  Mesh (
    ...
  )
  Container (
    VertexAttributeSetList (
      200 Include 4 10 21 22 11
    )
    Container (
      AttributeSet ( )
      SurfaceUV ( 0 0 )
    )
    Container (
      AttributeSet ( )
      SurfaceUV ( 0 1 )
    )
    Container (
      AttributeSet ( )
      SurfaceUV ( 1 1 )
    )
    Container (
      AttributeSet ( )
      SurfaceUV ( 1 0 )
    )
  )
)
```

● TRANSPARENCY COLOR

Creation Date 10/21/94
Mod Date 3/15/95

TYPE

Parent Hierarchy	Element, Attribute
Binary: kxpr 0x6B787072	**Ascii:** TransparencyColor

SIZE

12

PARENT OBJECTS

ALWAYS: AttributeSet

DATA FORMAT

```
ColorRGB  transparency
```

SUBOBJECTS

none

DESCRIPTION

The **transparency color** indicates the degree of light allowed to pass though the various channels (r,g,b) of a surface.

A color of (1, 1, 1) indicates complete transparency (meaning 100% of the light behind an object is allowed to pass through), a color of (0, 0, 0) indicates complete opacity (meaning no light passes through an object.)

EXAMPLE

```
Container (
  Polygon (
    ...
  )
  Container (
    AttributeSet ( )
    TransparencyColor ( 1 0 0 )
  )
)
```

● Generic Renderer

Creation Date 10/21/94
Mod Date 2/24/95

◁ ▷

Type

Parent Hierarchy Shared, Renderer

Referencable

Binary: gnrr 0x676E7272

Ascii: GenericRenderer

Size

0

Parent Objects

SOMETIMES: ViewHints

No Data

SubObjects

none

Description

A renderer that doesn't do anything, but may be used to accumulate state or for picking.

Example

```
Container (
  ViewHints ( )
  GenericRenderer ( )
  ViewAngleAspectCamera (
    ...
  )
  AmbientLight ( )
  Container (
    AttributeSet ( )
    DiffuseColor ( 0.2 0.2 0.2 )
  )
)
```

Interactive Renderer

Creation Date 10/21/94
Mod Date 4/6/95

Type

Parent Hierarchy Shared, Renderer — Referencable

Binary: ctwn 0x6374776E

Ascii: InteractiveRenderer

Size

0

Parent Objects

SOMETIMES: ViewHints

No Data

SubObjects

none

Description

The **interactive** renderer.

This will be renamed later when the corresponding product is named.

Example

```
Container (
  ViewHints ( )
  InteractiveRenderer ( )
  ViewAngleAspectCamera (
    ...
  )
  AmbientLight ( )
  Container (
    AttributeSet ( )
    DiffuseColor ( 0.2 0.2 0.2 )
  )
)
```

● *Wire Frame Renderer*

Creation Date 10/21/94
Mod Date 2/24/95

◁ ▷

Type	**Parent Hierarchy** Shared, Renderer — Referencable
	Binary: wrfr 0x77726672 — **Ascii:** WireFrame
Size	0
Parent Objects	SOMETIMES: ViewHints

No Data

SubObjects

none

Description

A **wireframe** renderer.

Example

```
Container (
  ViewHints ( )
  Wireframe ( )
  ViewAngleAspectCamera (
    ...
  )
  AmbientLight ( )
  Container (
    AttributeSet ( )
    DiffuseColor ( 0.2 0.2 0.2 )
  )
)
```

ATTRIBUTE SET

Creation Date 10/21/94
Mod Date 1/24/95

TYPE

Parent Hierarchy `Shared, Set` `Referencable`

Binary: `attr` `0x61747472`

Ascii: `AttributeSet`

SIZE

`0`

PARENT OBJECTS

`any AttributeSetList, any Geometry, any Group, any CapAttributeSet`

NO DATA

SUBOBJECTS

```
1 AmbientCoefficient (optional)
1 DiffuseColor (optional)
1 HighlightState (optional)
1 Normal (optional)
1 ShadingUV (optional)
1 SpecularColor (optional)
1 SpecularControl (optional)
1 SurfaceTangent (optional)
1 SurfaceUV (optional)
```

DESCRIPTION

A **attribute set** groups sets of unique attributes together and is associated with a vertex, face, or an entire geometry. Any object that is an **Element** may be placed in an **attribute set**.

An **attribute set** also may be placed in a group. The various attributes in an attribute set are inherited to nodes lower than it in a hierarchy.

EXAMPLE

```
Container (
  Mesh (
    ...
  )
  Container (
    VertexAttributeSetList (
      30 Exclude 2
      29 30
    )
    ...
    Container (
      AttributeSet ( )
      DiffuseColor ( 0 1 0 )
      SurfaceUV ( 0.87 0.57 )
    )
    ...
  )
)
```

ORTHOGRAPHIC CAMERA

Creation Date 10/21/94
Mod Date 3/15/95

TYPE

Parent Hierarchy Shared, Shape, Camera

Drawable Referencable

Binary: orth 0x6F727468

Ascii: OrthographicCamera

SIZE

16

PARENT OBJECTS

SOMETIMES: ViewHints

DATA FORMAT

```
Float32 left
Float32 top
Float32 right
Float32 bottom
```

- left < right
- bottom < top

SUBOBJECTS

```
1 CameraPlacement (optional, default)
1 CameraViewPort (optional, default)
1 CameraRange (optional, default)
```

DESCRIPTION

The lens characteristics are set with the dimensions of a rectangular view port in the frame of the camera.

EXAMPLE

```
OrthographicCamera (
  -1 -1 1 1
)

Container (
  OrthographicCamera (
    -1 -1 1 1
  )
  CameraPlacement (
    0 0 20
    0 0 0
    1 0 0
  )
  CameraRange (
    1 25
  )
)
```

View Angle Aspect Camera

Creation Date 10/21/94
Mod Date 1/24/95

TYPE	**Parent Hierarchy** Shared, Shape, Camera — Drawable Referencable
	Binary: vana 0x76616E61 — **Ascii:** ViewAngleAspectCamera
SIZE	8
PARENT OBJECTS	SOMETIMES: ViewHints

Data Format

```
Float32 fieldOfView
Float32 aspectRatioXtoY

• 0 < fieldOfView ≤ π
• 0 < aspectRatioXtoY
```

SubObjects

```
1 CameraPlacement (optional, default)
1 CameraViewPort (optional, default)
1 CameraRange (optional, default)
```

Description

A perspective camera specified in terms of the minimum view angle and the aspect ratio of X to Y.

Example

```
ViewAngleAspectCamera (
  1.7 1.0
)

Container (
  ViewAngleAspectCamera (
    1.7 1.0
  )
  CameraPlacement (
    0 0 20
    0 0 0
    1 0 0
  )
  CameraRange (
    1 25
  )
)
```

View Plane Camera

Creation Date 10/21/94
Mod Date 1/24/95

Type

Parent Hierarchy Shared, Shape, Camera

Drawable Referencable

Binary: vwpl 0x7677706C

Ascii: ViewPlaneCamera

Size

20

Parent Objects

SOMETIMES: ViewHints

Data Format

```
Float32 viewPlane
Float32 halfWidthAtViewPlane
Float32 halfHeightAtViewPlane
Float32 centerXOnViewPlane
Float32 centerYOnViewPlane
```

- 0 < viewPlane
- 0 < halfWidthAtViewPlane
- 0 < halfHeightAtViewPlane
- centerXOnViewPlane, centerYOnViewPlane may be any value

SubObjects

```
1 CameraPlacement (optional, default)
1 CameraViewPort (optional, default)
1 CameraRange (optional, default)
```

Description

A **view plane camera** is a **view angle aspect camera** specified in terms of an arbitrary view plane. This is most useful when setting the camera to look at a particular object.

The viewPlane is set to distance from the camera to the object.

The halfWidth is set to half the width of the cross section of the object, and the halfHeight equal to the halfWidth divided by the aspect ratio of the viewPort.

This is the only perspective camera with specifications for off-axis viewing, which is desirable for scrolling.

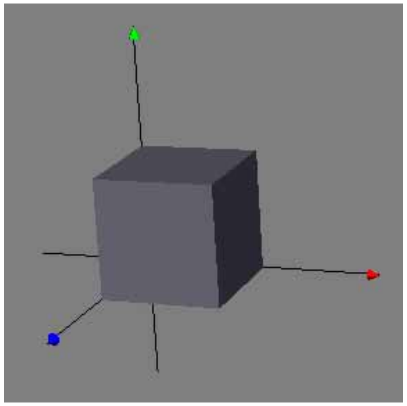
Example

```
ViewPlaneCamera (
  ...
)

Container (
  ViewPlaneCamera (
    20
    15.0 15.0
    18 29
  )
  CameraPlacement (
    0 0 20
    0 0 0
    1 0 0
  )
  CameraRange (
    1 25
  )
)
```

● *Box*

Creation Date 10/21/94
Mod Date 1/20/95

TYPE	**Parent Hierarchy** Shared, Shape, Geometry	Drawable Referencable
	Binary: box 0x626F7820	**Ascii:** Box
SIZE	0 or 48	
PARENT OBJECTS		



Data Format

```
Vector3D  orientation
Vector3D  majorAxis
Vector3D  minorAxis
Point3D   origin

• For 0-sized objects, default is:

1 0 0 # orientation
0 1 0 # majorAxis
0 0 1 # minorAxis
0 0 0 # origin
```

SubObjects

```
1 FaceAttributeSetList (optional, nObjects = 6)
1 AttributeSet (optional)
```

Description

This is a rectangular parallelepiped

A size of zero indicates the default values, helpful in instantiating a unit-cube.

The **Face Attribute Set List** subobject assigns color to the following faces:

Face ⊥ orientation at origin + orientation

Face ⊥ orientation at origin

Face ⊥ majorAxis at origin + majorAxis

Face ⊥ majorAxis at origin

Face ⊥ minorAxis at origin + minorAxis

Face ⊥ minorAxis at origin

Basically, the faces perpendicular to the “orientation” direction are assigned first, then the “majorAxis”, then the “minorAxis.”

Example

```
Box ( )

Box (
  2 0 0
  0 1 1
  2 3 0
  0 0 0
)

Container (
  Box ( )
  Container (
    FaceAttributeSetList (
      6 Exclude 0
    )
    Container (
      AttributeSet ( )
      DiffuseColor ( 1 0 0 )
    )
    Container (
      AttributeSet ( )
      DiffuseColor ( 0 1 1 )
    )
    Container (
      AttributeSet ( )
      DiffuseColor ( 0 1 0 )
    )
    Container (
      AttributeSet ( )
      DiffuseColor ( 1 0 1 )
    )
    Container (
      AttributeSet ( )
      DiffuseColor ( 0 0 1 )
    )
    Container (
      AttributeSet ( )
      DiffuseColor ( 1 1 0 )
    )
  )
)
```

Cone

Creation Date 10/21/94
Mod Date 1/14/95

Type

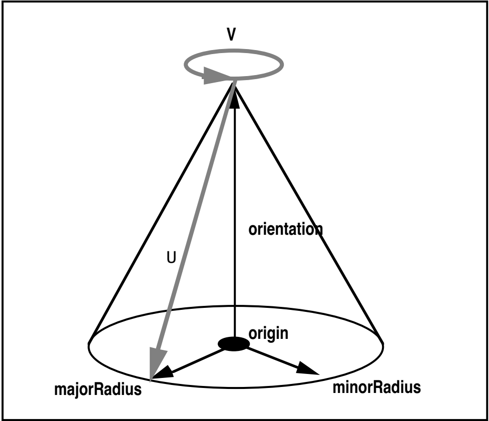
Parent Hierarchy Shared, Shape, Geometry — Drawable Referencable

Binary: cone 0x636F6E65 **Ascii:** Cone

Size

0 or 48

Parent Objects



Data Format

```
Vector3D  orientation
Vector3D  majorAxis
Vector3D  minorAxis
Point3D   origin

• For 0-sized objects, default is:

1 0 0 # orientation
0 1 0 # majorAxis
0 0 1 # minorAxis
0 0 0 # origin
```

SubObjects

```
1 Caps (optional, default)
1 FaceCapAttributeSet (optional)
1 BottomCapAttributeSet (optional)
1 AttributeSet (optional)
```

Description

A **cone** may have a cap, and may have attributes assigned to the entire **geometry**, to the "face" cap, or to the "bottom" cap.

The default parametrization is shown in the diagram.

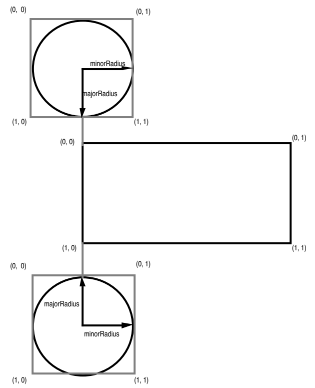
Example

```
Cone ( )

Cone (
  2 0 0
  0 1 1
  2 3 0
  0 0 0
)

Container (
  Cone ( )
  Caps ( Bottom )
  Container (
    BottomCapAttributeSet ( )
    Container (
      AttributeSet ( )
      DiffuseColor ( 1 0 0 )
    )
  )
  Container (
    FaceCapAttributeSet ( )
    Container (
      AttributeSet ( )
      DiffuseColor ( 1 1 0 )
    )
  )
)
```

CYLINDER

Creation Date 10/21/94
Mod Date 1/14/95

TYPE	**Parent Hierarchy** Shared, Shape, Geometry	Drawable Referencable
	Binary: cyln 0x63796C6E	**Ascii:** Cylinder
SIZE	0 or 48	
PARENT OBJECTS		



DATA FORMAT

```
Vector3D  orientation
Vector3D  majorRadius
Vector3D  minorRadius
Point3D   origin

• For 0-sized objects, default is:

1 0 0 # orientation
0 1 0 # majorAxis
0 0 1 # minorAxis
0 0 0 # origin
```

SUBOBJECTS

```
1 Caps (optional, default)
1 TopCapAttributeSet (optional)
1 FaceCapAttributeSet (optional)
1 BottomCapAttributeSet (optional)
1 AttributeSet (optional)
```

DESCRIPTION

A **cylinder** may have either top or bottom caps, and may have attributes assigned to the entire geometry, to the “face” cap, the “bottom” cap, or the “top” cap.

The default parametrization is shown in the diagram.

EXAMPLE

```
Cylinder ( )

Cylinder (
  2 0 0
  0 1 1
  2 3 0
  0 0 0
)

Container (
  Cylinder ( )
  Caps ( Bottom | Top )
  Container (
    BottomCapAttributeSet ( )
    Container (
      AttributeSet ( )
      DiffuseColor ( 0 1 0 )
    )
  )
  Container (
    FaceCapAttributeSet ( )
    Container (
      AttributeSet ( )
      DiffuseColor ( 1 0 1 )
    )
  )
  Container (
    TopCapAttributeSet ( )
    Container (
      AttributeSet ( )
      DiffuseColor ( 1 1 0 )
    )
  )
)
```

Disk

Creation Date 10/21/94
Mod Date 1/14/95

Type

Parent Hierarchy Shared, Shape, Geometry

Drawable Referencable

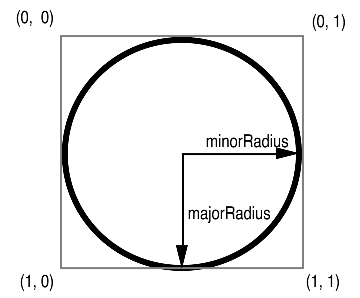
Binary: disk 0x6469736B

Ascii: Disk

Size

0 or 36

Parent Objects



Data Format

```
Vector3D  majorRadius
Vector3D  minorRadius
Point3D   origin

• For 0-sized objects, default is:

1 0 0 # majorRadius
0 1 0 # minorRadius
0 0 0 # origin
```

SubObjects

1 AttributeSet (optional)

Description

This is an elliptical **disk** at the given origin with two vectors specifying the dimensions.

The default parametrization is shown in the diagram.

Example

```
Disk ( )

Disk (
  2 0 0
  0 1 1
  0 0 0
)

Container (
  Cylinder ( )
  Caps ( Bottom | Top )
  Container (
    BottomCapAttributeSet ( )
    Container (
      AttributeSet ( )
      DiffuseColor ( 1 0 1 )
    )
  )
  Container (
    FaceCapAttributeSet ( )
    Container (
      AttributeSet ( )
      DiffuseColor ( 1 1 0 )
    )
  )
  Container (
    AttributeSet ( )
    DiffuseColor ( 1 1 0 )
  )
)
```

● ELLIPSE

Creation Date 10/21/94
Mod Date 1/14/95

TYPE

Parent Hierarchy Shared, Shape, Geometry

Drawable Referencable

Binary: elps 0x656C7073

Ascii: Ellipse

SIZE

0 or 36

PARENT OBJECTS

DATA FORMAT

```
Vector3D  majorAxis
Vector3D  minorAxis
Point3D   origin

• For 0-sized objects, default is:

1 0 0 # majorAxis
0 1 0 # minorAxis
0 0 0 # origin
```

SUBOBJECTS

```
1 AttributeSet (optional)
```

DESCRIPTION

This is an **ellipse** at the given origin with two vectors specifying its dimensions.

There is no default parametrization for an ellipse.

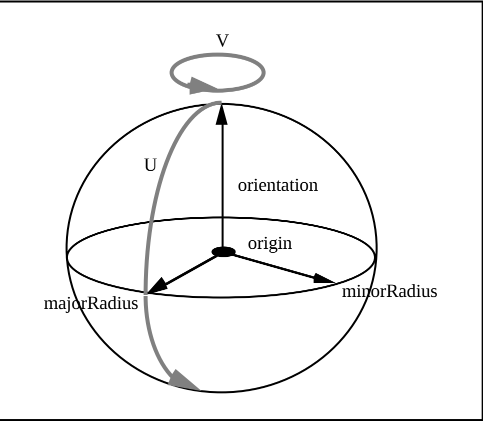
EXAMPLE

```
Ellipse ( )

Ellipse (
  2 0 0
  0 1 1
  0 0 0
)

Container (
  Ellipse ( )
  Container (
    AttributeSet ( )
    DiffuseColor ( 1 1 0 )
  )
)
```

ELLIPSOID

Creation Date 10/21/94
Mod Date 1/24/95

TYPE	**Parent Hierarchy** Shared, Shape, Geometry	Drawable Referencable
	Binary: elpd 0x656C7064	**Ascii:** Ellipsoid
SIZE	0 or 48	
PARENT OBJECTS		



DATA FORMAT

```
Vector3D  orientation
Vector3D  majorRadius
Vector3D  minorRadius
Point3D   origin

• For 0-sized objects, default is:

1 0 0 # orientation
0 1 0 # majorRadius
0 0 1 # minorRadius
0 0 0 # origin
```

SUBOBJECTS

1 AttributeSet (optional)

DESCRIPTION

An **ellipsoid** may have an attribute set attached to it.

The default parametrization is shown in the diagram. V is zero to the left of majorRadius, and is 1 to the right. U is zero at the orientation vector, and 0 at the bottom.

EXAMPLE

```
Sphere ( )

Sphere (
  2 0 0
  0 1 1
  2 3 0
  0 0 0
)

Container (
  Sphere ( )
  Container (
    AttributeSet ( )
    DiffuseColor ( 1 1 0 )
  )
)
```

General Polygon

Creation Date 10/21/94
Mod Date 4/7/95

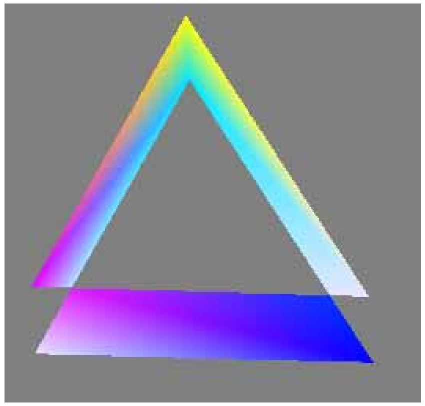
Type

Parent Hierarchy	Shared, Shape, Geometry	Drawable Referencable
Binary: gpgn 0x6770676E		**Ascii:** GeneralPolygon

Size

```
4 + sizeof(polygons[0..nCoutours-1])

sizeof(PolygonData) = 4 + nVertices * 12
```

Parent Objects



Data Format

```
Uns32        nContours
PolygonData  polygons[nContours]

where PolygonData is:

Uns32   nVertices
Point3D vertices[nVertices]

• 0 < nContours
• 2 < nVertices
```

SubObjects

```
1 VertexAttributeSetList (optional, nObjects =
nVertices[0] + ... + nVertices[nContours-1])
1 AttributeSet (optional)
1 GeneralPolygonHint (optional)
```

Description

A **general polygon** is a polygon that may be convex or may contain holes. A general polygon also assumes that all faces are planar within floating point tolerances.

Holes are indicated by specifying a contour of the generalPolygon in clockwise order.

Polygons that cross use the even-odd rule to specify holes (see diagram).

You may specify the complexity of a GeneralPolygon by adding a viewHints object.

Example

```
Container (
 GeneralPolygon (
  2 # nContours
  3 # nVertices
  -1 0 0
  1 0 0
  0 1.7 0
  3 # nVertices
  -1 0.4 0
  1 0.4 0
  0 2.1 0
 )
 Container (
  VertexAttributeSetList ( 6 Exclude 2 0 4 )
  Container (
   AttributeSet ( )
   DiffuseColor ( 0 0 1 )
  )
  Container (
   AttributeSet ( )
   DiffuseColor ( 0 1 1 )
  )
  Container (
   AttributeSet ( )
   DiffuseColor ( 1 0 1 )
  )
  Container (
   AttributeSet ( )
   DiffuseColor ( 1 1 0 )
  )
 )
 Container (
  AttributeSet ( )
  DiffuseColor ( 1 1 1 )
 )
)
```

Line

Creation Date 10/21/94
Mod Date 1/14/95

Type

Parent Hierarchy	Shared, Shape, Geometry	Drawable Referencable
Binary: line 0x6C696E65	**Ascii:** Line	

Size

24

Parent Objects

Data Format

```
Point3D start
Point3D end
```

SubObjects

```
1 VertexAttributeSetList (optional, nObjects = 2)
1 AttributeSet (optional)
```

Description

Our basic **line** primitive is a line segment, a simple line drawn between two vertices.

Optional vertex attributes may be attached using a **VertexAttributeSetList**.

A set of attributes may be applied to the entire line segment by attaching an **attribute set**.

Example

```
Line ( 0 0 0 1 0 0 )

Container (
  Line (
    0 0 0
    1 0 0
  )
  Container (
    VertexAttributeSetList ( 2 Exclude 0 )
    Container (
      AttributeSet ( )
      DiffuseColor ( 1 0 0 )
    )
    Container (
      AttributeSet ( )
      DiffuseColor ( 0 0 1 )
    )
  )
)
```

Marker

Creation Date 10/21/94
Mod Date 3/23/95

Type

Parent Hierarchy Shared, Shape, Geometry

Drawable Referencable

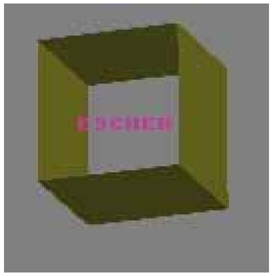
Binary: mrkr 0x6D726B72

Ascii: Marker

Size

32 + (rowBytes * height) + padding

Parent Objects



Data Format

```
Point3D    location
Int32      xOffset
Int32      yOffset
Uns32      width
Uns32      height
Uns32      rowBytes
EndianEnum bitEndian
RawData    data[height * rowBytes]
```

- 0 < width
- 0 < height
- (((width / 8) + ((width & 7) > 0)) ≤ rowBytes
- EndianEnum is:

Binary	Text
0x00000000	BigEndian
0x00000001	LittleEndian

SubObjects

1 AttributeSet (optional)

Description

The **marker** is used to rasterize bitmaps parallel to the viewing plane. They are used for annotation of an image.

Example

```
Container (
 Marker (
  0.5 0.5 0.5 # origin
  -28 # xOffset
  -3 # yOffset
  56 # width
  6 # height
  7 # rowBytes
  BigEndian # bitOrder
  0x7E3C3C667E7C18606066666066187C3C
  0x607E7C661860066066607C1860066666
  0x6066007E3C3C667E6618
 )
 Container (
  AttributeSet ( )
  DiffuseColor ( 0.8 0.2 0.6 )
 )
)
```

● *MESH*

Creation Date 10/21/94
Mod Date 3/15/95

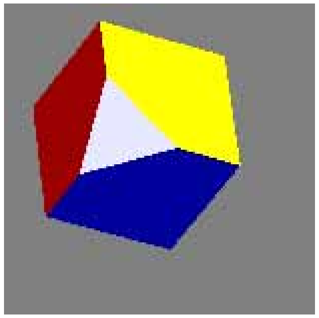
TYPE

Parent Hierarchy Shared, Shape, Geometry		Drawable Referencable
Binary: mesh	0x6D657368	**Ascii:** Mesh

SIZE

```
4 + nVertices * 12 +
  8 + (nFaces+nContours) * sizeof(faces[0..nFaces+nContours-1]

sizeof(MeshFace) = sizeof(Int) + sizeof(Uns) * nFaceVertexIndices
```

PARENT OBJECTS



DATA FORMAT

```
Uns32     nVertices
Point3D   vertices[nVertices]
Uns32     nFaces
Uns32     nContours
MeshFace  faces[nFaces + nContours]

• where MeshFace is:

Int32       nFaceVertexIndices
Uns32       faceVertexIndices[nFaceVertexIndices]

• 3 ≤ nVertices
• 3 ≤ nFaceVertexIndices
```

SUBOBJECTS

1 FaceAttributeSetList (optional, nObjects = nFaces)
1 VertexAttributeSetList (optional, nObjects = nVertices)
1 MeshCorners (optional)
1 AttributeSet (optional)

DESCRIPTION

The **mesh** is used for representing complex topological objects. It contains enough information to determine which polygonal faces are adjacent to each other without numerical ambiguity. This metafile object contains topological as well as geometrical information.

A contour (hole) in a face is indicated by supplying a negative number for the number of vertices, and adds a hole to the previous face that was not a contour.

The size of **nFaceVertexIndices** and **faceVertexIndices** is based on the value of **nVertices**.

We introduce a special subobject used only with the mesh, called “MeshCorners.” This object allows multiple attribute sets to be attached to a single vertex, where each attribute set is bound to a set of vertex-face pairs. It can be used to place a sharp edge in the mesh (if the attribute set contains a normal, for instance).

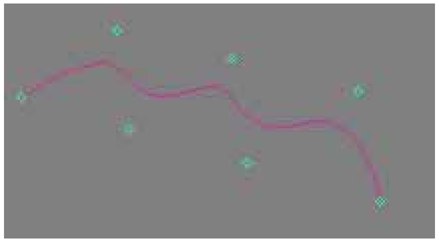
EXAMPLE

```
Mesh (
 10 # nVertices
 -1 1 1
 -1 1 -1
 1 1 -1
 1 -1 -1
 1 -1 1
 0 -1 1
 -1 -1 0
 -1 -1 -1
 1 1 1
 -1 0 1
 7 # nFaces
 0 # nContours
 3 6 5 9
 5 7 6 9 0 1
 4 2 3 7 1
 4 2 8 4 3
 4 1 0 8 2
 5 4 8 0 9 5
 5 3 4 5 6 7
)
```

● *Nurb Curve*

Creation Date 10/21/94
Mod Date 1/10/95

Type	**Parent Hierarchy** Shared, Shape, Geometry	Drawable Referencable
	Binary: nrbc 0x6E726263	**Ascii:** NURBCurve
Size	8 + (nPoints * 12) + ((nPoints + order) * 4)	
Parent Objects		



Data Format

```
Uns32           order
Uns32           nPoints
RationalPoint4D points[nPoints]
Float32         knots[order + nPoints]
```

- $2 \leq$ order
- $2 \leq$ nPoints
- $0 <$ points[...].w (weights of points)

SubObjects

none

Description

NURB curves are Non-Uniform Rational B-spline curves. A rational B-spline curve is a curve in 4D space, which has been projected down to 3D space. Thus, the control points for a 3D rational curve have four components - x, y, z, and w (usually known as the weight). For such a point, the corresponding point in 3D space is (x/w, y/w, z/w)

Weights (w) are always positive.

Example

```
NURBCurve (
 4 7 # order, nPoints
 0 0 0 1 # points
 1 1 0 1
 2 0 0 1
 3 1 0 1
 4 0 0 1
 5 1 0 1
 6 0 0 1
 0 0 0 0 0.25 0.5 0.75 1 1 1 1 # knots
)
```

NURB PATCH

Creation Date 10/21/94
Mod Date 3/20/95

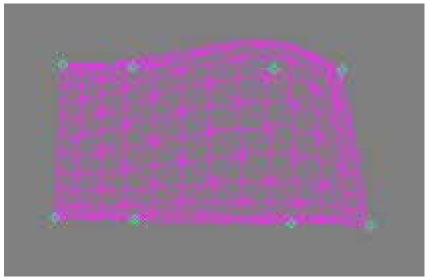
TYPE

Parent Hierarchy Shared, Shape, Geometry		Drawable Referencable
Binary: nrbp	0x6E726270	**Ascii:** NURBPatch

SIZE

```
16 + (16 * numColumns * numRows) + ((uOrder + numColumns) * 4) + ((vOrder +
numRows) * 4)
```

PARENT OBJECTS



DATA FORMAT

```
Uns32              uOrder
Uns32              vOrder
Uns32              numColumns
Uns32              numRows
RationalPoint4D    points[numMPoints*numNPoints]
Float32            uKnots[uOrder + numColumns]
Float32            vKnots[vOrder + numRows]
```

- 2 ≤ numColumns
- 2 ≤ numRows
- 2 ≤ uOrder
- 2 ≤ vOrder
- 0 < points[...].w (weights of points)

DESCRIPTION

Non-Uniform Rational B-Spline (NURB) Patches are closed under projective transformations, can represent quadrics exactly, and can be refined locally to allow additional detail.

The default parametrization is given by the knot vectors.

Weights (w) are always positive.

EXAMPLE

```
NURBPatch (
 4 4 4 4 # u,v order, num M,N points
 -2 2 0 1   -1 2 0 1   1 2 0 1   2 2 0 1
 -2 2 0 1   -1 2 0 1   1 0 5 1   2 2 0 1
 -2 -2 0 1  -1 -2 0 1  1 -2 0 1  2 -2 0 1
 -2 -2 0 1  -1 -2 0 1  1 -2 0 1  2 -2 0 1
  0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 # knots
)
```

SUBOBJECTS

1 TrimCurves (optional)

● POINT

Creation Date 10/21/94
Mod Date 1/24/95

TYPE	**Parent Hierarchy** Shared, Shape, Geometry — Drawable Referencable
	Binary: pnt 0x706E7420 — **Ascii:** Point
SIZE	12
PARENT OBJECTS	

DATA FORMAT

```
Point3D   point
```

SUBOBJECTS

```
1 AttributeSet (optional)
```

DESCRIPTION

The basic point primitive is an infinitesimally small point in space. It is specified as a 3D point plus an optional attribute set.

A 3D point has no default parametrization.

EXAMPLE

```
Point ( 0 1 2 )
```

POLYGON

Creation Date 10/21/94
Mod Date 1/14/95

TYPE	**Parent Hierarchy** `Shared, Shape, Geometry` — Drawable Referencable
	Binary: plyg 0x706C7967 — **Ascii:** Polygon
SIZE	`4 + nVertices * 12`
PARENT OBJECTS	

DATA FORMAT

```
Uns32   nVertices
Point3D vertices[nVertices]

• 2 ≤ nVertices
```

SUBOBJECTS

```
1 VertexAttributeSetList (optional, nObjects =
nVertices)
1 AttributeSet (optional)
```

DESCRIPTION

The **polygon** is convex with no holes. To describe concave polygons or polygons with holes, use the "**general polygon**" primitive.

The points that make up a **polygon**'s face are assumed to be planar within floating point tolerances.

EXAMPLE

```
Polygon (
  4
  0 1 1
  0 -1 1
  0 -1 -1
  0 1 -1
)
```

Poly Line

Creation Date 10/21/94
Mod Date 3/15/95

Type

Parent Hierarchy Shared, Shape, Geometry

Drawable Referencable

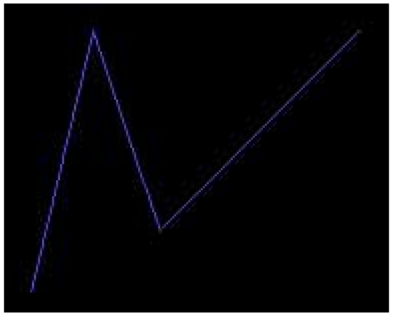
Binary: plyl 0x706C796C

Ascii: PolyLine

Size

4 + nVertices * 12

Parent Objects



Data Format

```
Uns32      nVertices
Point3D    vertices[nVertices]
```

• 2 ≤ nVertices

SubObjects

1 VertexAttributeSetList (optional, nObjects = nVertices)
1 GeometryAttributeSetList (optional, nObjects = nVertices - 1)
1 AttributeSet (optional)

Description

An extension of the basic line primitive is a **polyline**, where simple lines are drawn between adjacent points in a point list

A **polyline** is NOT closed, and the last point is never connected to the first point.

A **polyline** has no default parametrization.

Example

```
Container (
  PolyLine (
    4
    -1 -0.5 -0.25
    -0.5 1.5 0.45
    0 0 0
    1.5 1.5 1
  )
  Container (
    AttributeSet ( )
    DiffuseColor ( 0.4 0.2 0.9 )
  )
)
```

Torus

Creation Date 10/21/94
Mod Date 1/24/95

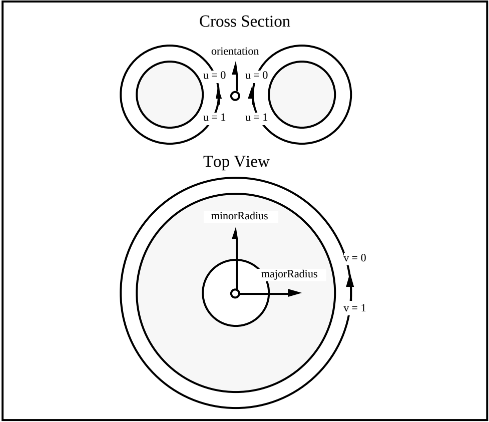
Type

Parent Hierarchy	Shared, Shape, Geometry	Drawable Referencable
Binary: tors 0x746F7273	**Ascii:** Torus	

Size

0 or 52

Parent Objects



Data Format

```
Vector3D  orientation
Vector3D  majorAxis
Vector3D  minorAxis
Point3D   origin
Float32   ratio
```

• For 0-sized objects, default is:

```
1 0 0 # orientation
0 1 0 # majorAxis
0 0 1 # minorAxis
0 0 0 # origin
1     # ratio
```

SubObjects

1 AttributeSet (optional)

Description

The orientation length specifies the radius of the circular along the orientation vector of the torus cross-section.

The major and minor axes are vectors to the center of the torus cross-section (as in the diagram).

The ratio is the change in the orientation length in the axial direction. A ratio of 2, for example, creates a fatter torus cross-section along the major and minor axes, a ratio of 0.5 creates a fatter cross-section along the orientation.

As far as anyone knows, the torus is useful for drawing donuts and bagels, and makes a great demo.

The default parametrization is shown in the diagram.

Example

```
Torus ( )

Torus (
  2 0 0
  0 1 1
  2 3 0
  0 0 0
  1
)

Container (
  Torus ( )
  Container (
    AttributeSet ( )
    DiffuseColor ( 1 1 0 )
  )
)
```

● TRIANGLE

Creation Date 10/21/94
Mod Date 1/14/95

TYPE

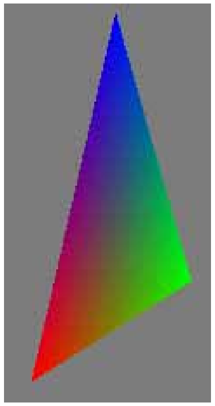
Parent Hierarchy Shared, Shape, Geometry   Drawable   Referencable

Binary: trng   0x74726E67   **Ascii:** Triangle

SIZE

36

PARENT OBJECTS



DATA FORMAT

```
Point3D  vertices[3]
```

SUBOBJECTS

```
1 VertexAttributeSetList (optional, nObjects = 3)
1 AttributeSet (optional)
```

DESCRIPTION

The most basic polygon is a **triangle**, which contains 3 points.

A **VertexAttributeSetList** may be used to attach attribute sets to the vertices (containing three vertex attribute sets) or an optional **AttributeSet** may be added to attach to the face.

There is no default parametrization for a triangle.

EXAMPLE

```
Container (
 Triangle (
  -1 -0.5 -0.25
  0 0 0
  -0.5 1.5 0.45
 )
 Container (
  VertexAttributeSetList ( 3 Exclude 0 )
  Container (
   AttributeSet ( )
   DiffuseColor ( 1 0 0 )
  )
  Container (
   AttributeSet ( )
   DiffuseColor ( 0 1 0 )
  )
  Container (
   AttributeSet ( )
   DiffuseColor ( 0 0 1 )
  )
 )
 Container (
  AttributeSet ( )
  DiffuseColor ( 0.8 0.5 0.2 )
 )
)
```

TRI GRID

Creation Date 10/21/94
Mod Date 3/20/95

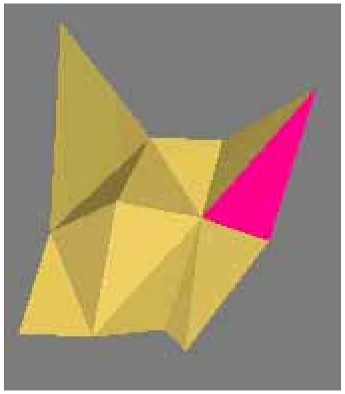
TYPE

Parent Hierarchy	Shared, Shape, Geometry — Drawable Referencable
Binary: trig 0x74726967	**Ascii:** TriGrid

SIZE

8 + (nColumns * nRows * 12)

PARENT OBJECTS



DATA FORMAT

```
Uns32     nColumns
Uns32     nRows
Point3D   points[numMVertices * numNVertices]
```

• 2 ≥ nColumns
• 2 ≥ nRows

SUBOBJECTS

1 FaceAttributeSetList (optional, nObjects = (numNVertices - 1) * (numMVertices - 1) * 2)
1 VertexAttributeSetList (optional, nObjects = numNVertices * numMVertices attribute sets)
1 AttributeSet (optional)

DESCRIPTION

Points specified are given in row major order.

You may add a **FaceAttributeSetList** to attach a set of attributes for each of the triangles generated by this primitive.

You may also add a **VertexAttributeSetList** to attach attributes to each vertex.

EXAMPLE

```
Container (
 TriGrid (
  3 4 # nUVertices nVVertices
  -1 1 1      -0.5 1 0  0 1 0
   0.7 1 0.5  -1 0 0    -0.5 0 0.3
   0 0.2 0    0.5 0 0   -1 -1 0
   -0.5 -1 0  0 -1 0.1  0.2 -1.3 0.2
 )
 Container (
  FaceAttributeSetList ( 12 Include 1 5 )
  Container (
   AttributeSet ( )
   DiffuseColor ( 1 0 0.5 )
  )
 )
 Container (
  AttributeSet ( )
  DiffuseColor ( 0.8 0.7 0.3 )
 )
)
```

Group

Creation Date 10/26/94
Mod Date 3/8/95

Type

Parent Hierarchy	Shared, Shape — Drawable Referencable
Binary: grup 0x67727570	**Ascii:** Group

Size

0

Parent Objects

none

No Data

SubObjects

none

Description

The **group** is useful for grouping any type of shared objects together.

It is delimited by an **end group** object.

Example

```
BeginGroup ( Group ( ) )
  CString ( “This is the first day of the rest
of your life.” )
  Torus ( )
EndGroup ( )
```

● Display Group

Creation Date 10/21/94
Mod Date 1/24/95

Type	**Parent Hierarchy** Shared, Shape, Group — Drawable Referencable
	Binary: dspg 0x6C697374 — **Ascii:** DisplayGroup
Size	0
Parent Objects	

No Data

SubObjects

1 DisplayGroupState (optional)

Description

A **display group** contains only objects that are drawable.

A **display group** adds the ability to be traversed for various operations via the **DisplayGroupState** subobject.

It is delimited by an **end group** object.

Example

Io Proxy Display Group

Creation Date 1/24/95
Mod Date 4/6/95

Type

Parent Hierarchy Shared, Shape, Group, DisplayGroup

Drawable Referencable

Binary: iopx 0x70727879

Ascii: IOProxyDisplayGroup

Size

0

Parent Objects

No Data

SubObjects

1 DisplayGroupState (optional, default)

Description

The **IO proxy display group** contains drawable objects that are similar representations of the same object in different formats. For example, if it is known that a particular application does not understand NURBPatch's, the writing application may write the NURBPatch in an IO proxy group along with a mesh which is the tesselated NURBPatch.

The objects in a **IO proxy display group** appear in their preferencial order. The first object is the most preferred representation, the last object the least. The first object that is "understood" by a reading application should be used.

You may specify a group of objects inside a **IOProxyDisplayGroup**, as a group (up to its "**EndGroup**") delimiter is a single object.

It is understood that ONLY the first understood object in an **IO proxy display group** is traversed while drawing, bounding, or picking.

In other words, if an IO proxy display group contains many objects, only one of them will be drawn when it comes time to render an image, etc.

Example

```
BeginGroup ( IOProxyDisplayGroup ( ) )
  Mesh (
    8
    0 0 0
    0 0 1
    0 1 0
    1 0 0
    1 1 0
    0 1 1
    1 0 1
    1 1 1
    ... etc.
  )
  Box ( )
EndGroup ( )

BeginGroup ( IOProxyDisplayGroup ( ) )
  NURBPatch (       # preferred object
    ...
  )
  DisplayGroup ( )  # 2nd choice object
    Translate ( 1 2 3 )
    Box ( )
  EndGroup ( )
EndGroup ( )
```

Ordered Display Group

Creation Date 10/21/94
Mod Date 1/24/95

Type

Parent Hierarchy	`Shared, Shape, Group, DisplayGroup` — `Drawable` `Referencable`
Binary: `ordg` `0x6F72646C`	**Ascii:** `OrderedDisplayGroup`

Size

`0`

Parent Objects

No Data

SubObjects

`1 DisplayGroupState (optional, default)`

Description

The **ordered display group** is simply a **display group** except that objects are sorted by type. Objects always appear in an **ordered group** in the following order:

- Transforms
- Styles
- AttributeSets
- Shaders
- Geometries
- DisplayGroups

It is delimited by an **end group** object.

Example

● Info Group

Creation Date 1/24/95
Mod Date 2/24/95

◁ ▷

Type

Parent Hierarchy `Shared, Shape, Group`

`Drawable` `Referencable`

Binary: `info` `0x696E666F`

Ascii: `InfoGroup`

Size

`0`

Parent Objects

`none`

No Data

SubObjects

`none`

Description

An **info group** contains nothing but **String** objects. It is used to add human-readable information pertaining to a file's origin or history. A use that comes to mind is copyright notices.

The **info group** object should be preserved by a reading application, and appended with additional information if a file is re-written.

It is delimited by an **end group** object.

Example

```
BeginGroup ( InfoGroup ( ) )
  CString (
    “Copyright © 1995 Apple Computer, Inc.” )
  CString (
    “Author: Bonanza Jellybean” )
EndGroup ( )
```

● Light Group

Creation Date 10/21/94
Mod Date 2/24/95

Type

Parent Hierarchy	Shared, Shape, Group	Drawable Referencable
Binary: lghg 0x676C6768	**Ascii:** LightGroup	

Size

0

Parent Objects

none

No Data

SubObjects

none

Description

A **light group** contains nothing but **lights**.

It is delimited by an **end group** object.

Example

```
BeginGroup ( LightGroup ( ) )
  AmbientLight ( )
  DirectionalLight ( 1 0 0 False )
EndGroup ( )
```

Ambient Light

Creation Date 10/21/94
Mod Date 10/27/94

Type

Parent Hierarchy Shared, Shape, Light

Drawable Referencable

Binary: ambn 0x616D626E

Ascii: AmbientLight

Size

0

Parent Objects

No Data

SubObjects

1 LightData (optional, default)

Description

An **ambient light** supplies light that comes from secondary reflections.

In lieu of other light sources, the **ambient light** illuminates the scene with a flat, uniform light.

Example

```
AmbientLight ( )

Container (
  AmbientLight ( )
  LightData (
    EcTrue # isOn
    1.0 # intensity
    1 0 0 # red color
  )
)
```

● Directional Light

Creation Date	10/21/94
Mod Date	3/15/95

Type

Parent Hierarchy Shared, Shape, Light   Drawable Referencable

Binary: drct 0x64726374   **Ascii:** DirectionalLight

Size

Parent Objects

Data Format

```
Vector3D    direction
Boolean     castsShadows
```

• |direction| = 1.0

SubObjects

1 LightData (optional, defaults)

Description

A **directional light** is far enough away from the scene that we may treat it as though it were infinitely far away. This produces shading results faster than any other type of light (except ambient).

It is specified with a vector pointing in the same direction as the light rays, an attenuation and a boolean value indicating whether this light casts shadows or not.

Example

```
DirectionalLight ( 1 0 0 True )

Container (
  DirectionalLight ( 1 0 0 True )
  LightData (
    True
    0.4
    1 0 0
  )
)
```

Point Light

Creation Date 10/21/94
Mod Date 3/15/95

Type

Parent Hierarchy	Shared, Shape, Light — Drawable Referencable
Binary: pntl 0x706E746C	**Ascii:** PointLight

Size

Parent Objects

Data Format

```
Point3D      location
Attenuation  attenuation
Boolean      castsShadows
```

• where **Attenuation** is the structure:

```
Float32  c0
Float32  c1
Float32  c2
```

• **attenuation** is computed, using **d** as the distance from **location**:

$$\frac{1}{c0 + c1*d + c2 * d^2}$$

• $0 < c0$
• $0 < c1$
• $0 < c2$

• **attenuation** is not clamped to [0,1] to allow for lighting washout (such as in a nuclear explosion)

SubObjects

1 LightData (optional, defaults)

Description

A **point light** is a light at an infinitesimally small point in space. It may be attenuated or it may cast shadows.

Example

```
PointLight (
  12 23 2
  0 0 1 # InverseDistanceSquared
  True
)

Container (
  PointLight (
    12 23 2
    0 0 1 # InverseDistanceSquared
    True
  )
  LightData (
    True
    0.4
    1 0 0
  )
)
```

Spot Light

Creation Date 10/21/94
Mod Date 1/24/95

Type

Parent Hierarchy	Shared, Shape, Light — Drawable Referencable
Binary: spot 0x73706F74	**Ascii:** SpotLight

Size

Parent Objects

Data Format

```
Point3D      location
Vector3D     orientation
Boolean      castsShadows
Attenuation  attenuation
Float32      hotAngle
Float32      outerAngle
FallOffEnum  fallOff
```

• |orientation| = 1

• Attenuation is described in the Point Light

• $0 < \text{hotAngle} \le \text{outerAngle} \le \pi$

• FallOffEnum is:

Binary	Text
0x00000000	None
0x00000001	Linear
0x00000002	Exponential
0x00000003	Cosine

SubObjects

1 LightData (optional, defaults)

Description

A **spot light** radiates with a circular cone of light that tapers toward the edge of the cone.

The hotSpotAngle is the angle (in radians) from the axis of the spot light for which the spot light has maximum, constant intensity. The outer angle is the angle for which the light falls to zero. Between these two, the light intensity tapers to zero according to the "FallOff" enumerated type.

Example

```
SpotLight (
  12 0 0
  0 1 0
  True
  0 0 1 # InverseDistanceSquared
  0.7 # hotAngle
  0.8 # outerAngle
  Cosine
)

Container (
  SpotLight (
    12 0 0
    0 1 0
    True
    0 0 1 # InverseDistanceSquared
    0.7 # hotAngle
    0.8 # outerAngle
   Cosine
  )
  LightData (
    True
    0.4
    1 0 1
  )
)
```

● Lambert Illumination

Creation Date 10/21/94
Mod Date 10/27/94

Type	**Parent Hierarchy** Shared, Shape, Shader, IlluminationShader	Inherited: Drawable Referencable
	Binary: `lmil` 0x6C6D696C	**Ascii:** `LambertIllumination`
Size	0	
Parent Objects		

No Data

SubObjects

none

Description

The lambertian illumination model.

Example

```
LambertIllumination ( )
```

● Phong Illumination

Creation Date 10/21/94
Mod Date 10/27/94

◁ ▷

Type

Parent Hierarchy	Shared, Shape, Shader, IlluminationShader	Inherited: Drawable Referencable
Binary: phil 0x7068696C	**Ascii:** PhongIllumination	

Size

0

Parent Objects

No Data

SubObjects

Description

The phong illumination model.

Example

```
PhongIllumination ( )
```

Texture Shader

Creation Date 10/21/94
Mod Date 10/27/94

Type

Parent Hierarchy	Shared, Shape, Shader, SurfaceShader	Inherited: Drawable Referencable
Binary: txsu 0x74787375	**Ascii:** TextureShader	

Size

0

Parent Objects

No Data

SubObjects

1 PixmapTexture (required)

Description

The **texture shader** is used to perform shading using a texture (in this case, a PixmapTexture).

Example

```
Container (
  TextureShader ( )
  PixmapTexture (
    ...
  )
)
```

Backfacing Style

Creation Date 10/21/94
Mod Date 10/27/94

Type	**Parent Hierarchy** Shared, Shape, Style	Inherited: Drawable Referencable
	Binary: bckf 0x62636B66	**Ascii:** BackfacingStyle
Size		
Parent Objects		

Data Format

```
BackfacingEnum  backfacing

• where BackfacingEnum is:

Text        Binary
0x00000000  Both
0x00000001  Culled
0x00000002  Flipped
```

SubObjects

none

Description

The **backfacing style** tells a renderer how to clip backfacing polygons while rendering.

Example

```
BackfacingStyle ( Culled )
```

● *Fill Style*

Creation Date 10/21/94
Mod Date 1/10/95

Type

Parent Hierarchy Shared, Shape, Style

Inherited
Drawable Referencable

Binary: fist 0x66697374

Ascii: FillStyle

Size

4

Parent Objects

Data Format

```
FillStyleEnum     fillStyle

• where FillStyleEnum is:
```

Text	Binary
0x00000000	Filled
0x00000001	Edges
0x00000002	Points
0x00000003	Empty

SubObjects

none

Description

The **fill style** tells a renderer what parts of a polygon to draw.

Example

```
FillStyle ( Edges )
```

Highlight Style

Creation Date 10/21/94
Mod Date 3/8/95

Type

Parent Hierarchy Shared, Shape, Style	Inherited: Drawable Referencable
Binary: high 0x68696768	**Ascii:** HighlightStyle

Size

0

Parent Objects

No Data

SubObjects

1 AttributeSet (required)

Description

The **highlight style** sets the binding for highlighting features of a geometry via the **HighlightState** attribute. The **attribute set** subobject sets the highlight attribute set.

Example

```
Container (
  HighlightStyle ( )
  Container (
    AttributeSet ( )
    DiffuseColor ( 0 0 1 )
  )
)
```

Interpolation Style

Creation Date 10/21/94
Mod Date 10/27/94

Type

Parent Hierarchy Shared, Shape, Style

Inherited
Drawable Referencable

Binary: intp 0x696E7470

Ascii: InterpolationStyle

Size

4

Parent Objects

Data Format

```
InterpolationStyleEnum     interpolationStyle
```

• where InterpolationStyleEnum is:

Binary	Text
0x00000000	None
0x00000001	Vertex
0x00000002	Pixel

SubObjects

none

Description

The **interpolation style** tells a renderer how to interpolate shading values on a polygon.

Example

```
InterpolationStyle ( Vertex )
```

Orientation Style

Creation Date 10/21/94
Mod Date 3/8/95

Type

Parent Hierarchy Shared, Shape, Style

Inherited
Drawable Referencable

Binary: ornt 0x6F726E74

Ascii: OrientationStyle

Size

4

Parent Objects

Data Format

```
OrientationEnum orientation

• where OrientationEnum is:
```

Binary	Text
0x00000000	CounterClockwise
0x00000001	Clockwise

Description

The Orientation style is used to change the orientation of polygons.

Example

```
OrientationStyle ( Clockwise )
```

SubObjects

none

● PICK ID STYLE

Creation Date 10/21/94
Mod Date 10/27/94

TYPE	**Parent Hierarchy** Shared, Shape, Style	Inherited Drawable Referencable
	Binary: pkid 0x706B6964	**Ascii:** PickIDStyle
SIZE	4	
PARENT OBJECTS		

DATA FORMAT

```
Uns32   id
```

SUBOBJECTS

none

DESCRIPTION

The **pick ID style** is used to allow the user to insert ids within a hierarchy to aid in picking a hierarchy.

EXAMPLE

```
PickIDStyle ( 23 )
```

PICK PARTS STYLE

Creation Date 10/21/94
Mod Date 3/8/95

TYPE

Parent Hierarchy Shared, Shape, Style

Inherited
Drawable Referencable

Binary: pkpt 0x706B7074

Ascii: PickPartsStyle

SIZE

PARENT OBJECTS

DATA FORMAT

```
PickPartsFlags    pickParts

• where PickPartsFlags is:

Text         Binary
0x00000000   Object
0x00000001   Face
0x00000002   Edge
0x00000004   Vertex

• default is:
  Object
```

SUBOBJECTS

none

DESCRIPTION

The **pick parts style** determines the level of granularity for picking.

EXAMPLE

```
PickPartsStyle ( Object | Vertex )
```

Receive Shadows Style

Creation Date 10/21/94
Mod Date 10/27/94

Type

Parent Hierarchy	Shared, Shape, Style	Inherited: Drawable Referencable
Binary: rcsh 0x72637368	**Ascii:** ReceiveShadowsStyle	

Size

4

Parent Objects

Data Format

```
Boolean  receiveShadows
```

SubObjects

none

Description

The **receive shadows style** determines whether a geometry receives shadows when rendering. It is coupled with the “casts shadows” field in all lights, excluding the ambient light.

Example

```
ReceiveShadowsStyle ( True )
```

Subdivision Style

Creation Date 10/21/94
Mod Date 3/15/95

Type

Parent Hierarchy	Shared, Shape, Style	Inherited: Drawable Referencable
Binary: sbdv 0x7364636C		**Ascii:** SubdivisionStyle

Size

```
(subdivisionMethod == Constant) ? 12 : 8
```

Parent Objects

Data Format

This object has two forms, based on the subdivison method field:

• for subdivisionMethod == WorldSpace or ScreenSpace the structure is:

```
SubdivisionMethodEnum subdivisionMethod
Float32               value1
```

• for subdivisionMethod == Constant, the values are integral:

```
SubdivisionMethodEnum subdivisionMethod
Uns32                 value1
Uns32                 value2
```

where SubdivisionMethodEnum is:

Binary	Text
0x00000000	Constant
0x00000001	WorldSpace
0x00000002	ScreenSpace

SubObjects

none

Description

The **subdivision style** tells a geometric decomposition the courseness of a geometric primitive tesselation. There are three methods of subdivision: constant, world space, and screen space subdivision.

Constant subdivision supplies 2 integral values, which indicate the number of sections the u and v axes of a decomposition should be divided into.

The Screen Space value indicates average size of a single polygon in a tesselation in screen space.

The world space value indicates the average size of a single polygon in a tesselation in world space.

Example

```
SubdivisionStyle (
   Constant 12 12
)

SubdivisionStyle (
   WorldSpace 50
)

SubdivisionStyle (
   ScreenSpace 50
)
```

Matrix Transform

Creation Date 10/21/94
Mod Date 10/27/94

Type	**Parent Hierarchy** Shared, Shape, Transform — Inherited: Drawable Referencable
	Binary: mtrx 0x6D747278 — **Ascii:** Matrix
Size	64
Parent Objects	

Data Format

```
Matrix4x4  matrix
```

• matrix is invertible

Description

A custom, invertible matrix transform.

Example

SubObjects

none

● Quaternion Transform

Creation Date 10/21/94
Mod Date 1/24/95

Type

Parent Hierarchy	Shared, Shape, Transform	Inherited: Drawable	Referencable
Binary: qtrn	0x7174726E	**Ascii:** Quaternion	

Size

16

Parent Objects

Data Format

```
Float32   w
Float32   x
Float32   y
Float32   z
```

Description

The quaternion specifies three axes of rotation and a "twist" value.

Useful for user interface.

Example

```
Quaternion ( 0.2 0.7 0.2 1.57 )
```

SubObjects

none

Rotate Transform

Creation Date 10/21/94
Mod Date 10/27/94

Type	**Parent Hierarchy** Shared, Shape, Transform	Inherited: Drawable Referencable
	Binary: rott 0x726F7474	**Ascii:** Rotate
Size		
Parent Objects		

Data Format

```
AxisEnum    axis
Float32     radians

• AxisEnum is:

Binary        Text
0x00000000    X
0x00000001    Y
0x00000002    Z
```

Description

Rotate about the X, Y, or Z axes.

Example

```
Rotate ( X 1.57 )
```

SubObjects

none

Rotate About Axis Transform

Creation Date 10/21/94
Mod Date 10/27/94

Type

Parent Hierarchy Shared, Shape, Transform

Inherited: Drawable Referencable

Binary: rtaa 0x72746161

Ascii: RotateAboutAxis

Size

28

Parent Objects

Data Format

```
Point3D  origin
Vector3D orientation
Float32  radians

• |orientation| = 1
```

Description

Rotate about an arbitrary axis in space.

Example

```
RotateAboutAxis (
  20 0 0 # origin
  0  1 0 # orientation
  1.57 # radians
)
```

SubObjects

none

Rotate About Point Transform

Creation Date 10/21/94
Mod Date 10/27/94

Type	**Parent Hierarchy** Shared, Shape, Transform		Inherited: Drawable Referencable
	Binary: rtap 0x72746170	**Ascii:** RotateAboutPoint	
Size	20		
Parent Objects			

Data Format

```
AxisEnum   axis
Float32    radians
Point3D    origin
```

• AxisEnum is:

Binary	Text
0x00000000	X
0x00000001	Y
0x00000002	Z

Description

To rotate about the X, Y, or Z axes at an arbitrary point in space.

Example

SubObjects

none

Scale Transform

Creation Date 10/21/94
Mod Date 10/27/94

Type	**Parent Hierarchy** Shared, Shape, Transform	Inherited: Drawable Referencable
	Binary: scal 0x7363616C	**Ascii:** Scale
Size		
Parent Objects		

Data Format	Description	Example
`Vector3D scale`	A scale transform.	`Scale ( 1 1 2 )`

SubObjects

none

Translate Transform

Creation Date 10/21/94
Mod Date 10/27/94

Type	**Parent Hierarchy** Shared, Shape, Transform	Inherited: Drawable Referencable
	Binary: trns 0x74726E73	**Ascii:** Translate
Size	12	
Parent Objects		

Data Format

```
Vector3D     translate
```

Description

A translate transfrom.

Example

```
Translate ( 1 2 100 )
```

SubObjects

none

Unknown Binary

Creation Date 4/6/95
Mod Date 4/7/95

Type

Parent Hierarchy	Shared, Shape — Drawable Referencable
Binary: ukbn 0x756B626E	**Ascii:** UnknownBinary

Size

12 +

Parent Objects

Data Format

```
Int32      objectType
Uns32      objectSize
EndianEnum byteOrder
RawData    objectData[objectSize]
```

SubObjects

Description

The unknown binary object is a way of transporting unknown data found in a binary file. It is an encapsulated replica of the original data found in a binary metafile, containing the object type (an Int32), the object size (in bytes), the byte order of the original file, and the data itself. The byte order is needed if unknown data is transported across different processors, and allows for parsing endian-specific primitives within the raw data block.

Unknown binary objects may be written in either the text or binary files.

When an unknown binary object is encountered in a metafile, it is up to the reading program to either:
• transport the data around
• validate it and convert it to a known object
• discard the data

Unknown objects are inherently “dirty”, meaning you may assume the unknown binary object may contain out-of-sync (bogus) information, as the original object may have been removed from its original context.

Example

```
UnknownBinary (
  1701605476
  4
  BigEndian
  0x0AB2
)
```

Unknown Text

Creation Date 4/6/95
Mod Date 4/7/95

Type	**Parent Hierarchy** Shared, Shape — Drawable Referencable
	Binary: uktx 0x756B7478 — **Ascii:** UnknownText
Size	sizeof(name) + sizeof(data)
Parent Objects	any

Data Format

```
String  asciiName
String  contents
```

SubObjects

Description

The unknown text object is a way of transporting unknown data found in a text file. It is an encapsulated replica of the original data found in a text metafile, containing the object type (a String), and a text string containing the original data. In some cases, white space and comments may have been stripped from the contents field.

Unknown text objects may be written in either the text or binary files.

When an unknown text object is encountered in a metafile, it is up to the reading program to either:
• transport the data around
• validate it and convert it to a known object
• discard the data

Unknown objects are inherently “dirty”, meaning you may assume the unknown text object may contain out-of-sync (bogus) information, as the original object may have been removed from its original context.

Example

```
UnknownText (
  “Ellipsoid”
  “”
)
```

Macintosh Path

Creation Date 10/21/94
Mod Date 3/15/95

Type	**Parent Hierarchy** Shared, Storage — Referencable
	Binary: macp 0x6D616370 — **Ascii:** MacintoshPath
Size	sizeof(String)
Parent Objects	ALWAYS: Reference

Data Format

```
String  pathName
```

SubObjects

none

Description

The Macintosh path specifies the pathname of an external file reference using the pathname specification found in the Inside Macintosh volumes. (essentially, a colon-based separator)

Example

```
Container (
  Reference ( 43 )
  MacintoshPath ( “:::Foo:Bar:Models:Cheryl” )
)
```

Unix Path

Creation Date 10/21/94
Mod Date 3/8/95

Type	**Parent Hierarchy** Shared, Storage	Referencable
	Binary: unix 0x756E6978	**Ascii:** UnixPath
Size	sizeof(String)	
Parent Objects	ALWAYS: Reference	

Data Format

```
String  unixPath
```

SubObjects

none

Description

The unix path object serves as a way to reference files on a unix file system.

The path should obey naming standards for unix operating systems.

Example

```
Container (
  Reference ( 23 )
  UnixPath ( “./shaders.eb” )
)
```

C String

Creation Date 10/21/94
Mod Date 3/15/95

Type

Parent Hierarchy `Shared, String` `Referencable`

Binary: `strc` `0x73747263` **Ascii:** `CString`

Size

```
sizeof(String)
```

Parent Objects

Data Format

```
String   cString
```

SubObjects

```
none
```

Description

The CString is a way of embedding text in a metafile.

Other string types allow for internationalization.

The only allowable characters in a CString are 7-bit ASCII numbers.

The following characters may be “escaped” with the ‘\’:

Example

```
CString (
  “Copyright (c) 1994 Apple Computer, Inc.”
)
```

● UNICODE

Creation Date 10/21/94
Mod Date 3/15/95

TYPE

Parent Hierarchy Shared, String

Referencable

Binary: uncd 0x756E6364

Ascii: Unicode

SIZE

```
4 + length * 2
```

PARENT OBJECTS

DATA FORMAT

```
Uns32    length
RawData  unicode[length * 2]
```

SUBOBJECTS

none

DESCRIPTION

The unicode object is another way of embedding text in a metafile.

See UNICODE reference for details.

EXAMPLE

```
Unicode (
  6
  0x457363686572
)
```

Pixmap Texture

Creation Date 10/21/94
Mod Date 1/24/95

Type

Parent Hierarchy	Shared, Texture — Referencable
Binary: txpm 0x7478706D	**Ascii:** PixmapTexture

Size

28 + rowBytes * height + padding

Parent Objects

SOMETIMES: TextureShader

Data Format

```
Uns32         width
Uns32         height
Uns32         rowBytes
Uns32         pixelSize
PixelTypeEnum pixelType
EndianEnum    bitOrder
EndianEnum    byteOrder
RawData       image[rowBytes * height]
```

• 0 < width
• 0 < height
• 0 < pixelSize < 32
• width * pixelSize ≤ rowBytes
• PixelTypeEnum is:

Binary	Text
0x00000000	RGB8
0x00000001	RGB16
0x00000002	RGB24
0x00000003	RGB32

• EndianEnum is:

Binary	Text
0x00000000	BigEndian
0x00000001	LittleEndian

SubObjects

none

Description

A generic means of transferring pixmap data. Used in the Texture Shader.

Example

```
PixmapTexture (
   256 256 # width/height
   128 # rowBytes
   32 # pixelSize
   RGB24
   BigEndian BigEndian
   0x00123232...
   0x...
)
```

VIEW HINTS

Creation Date 1/20/95
Mod Date 2/24/95

TYPE

Parent Hierarchy Shared		Referencable
Binary: vwhn	0x7677686E	**Ascii:** ViewHints

SIZE

0

PARENT OBJECTS

none

NO DATA

SUBOBJECTS

```
1 Renderer (optional)
1 Camera (optional)
many Lights (optional)
1 AttributeSet (optional)
1 ImageDimensions (optional)
1 ImageMask (optional)
1 ImageClearColor (optional)
```

DESCRIPTION

The subobjects of the **view hints** object specifies the preferences supplied by a writing application when rendering a scene.

The semantic to be followed when a **view hints** object is encountered in the metafile is that the view hints is specified previous to a list of objects to be rendered to that particular view hints preference. The subobjects of the view hints object are inherited from the previous view hints in a metafile.

For example, if a modelling application contains 10 camera locations for viewing various portions of a scene, it would first store the default view as the first object in a metafile, then the group representing the scene, then a view containing the second camera position, then a reference to the scene, etc.

EXAMPLE

```
3DMetafile ( 1 0 Normal toc> )
Container (
  ViewHints ( )
  Container (
    ViewAngleAspect ( 0.73 1.0 )
    CameraPlacement (
      0 0 30
      0 0 0
      0 1 0
    )
  )
  DirectionalLight ( -0.7 -0.7 -0.65 )
  Container (
    AttributeSet ( )
    DiffuseColor ( 0.2 0.2 0.2 )
    SpecularControl ( 3 )
  )
  ImageDimensions ( 200 200 )
)
ref1:
BeginGroup ( DisplayGroup ( ) )
...
EndGroup ( )
Container (
  ViewHints ( )
  Container (
    ViewAngleAspect ( 0.73 1.0 )
    CameraPlacement (
      0 10 0
      0 0 0
      0 1 0
    )
  )
)
Reference ( 1 )
```

PT 24 - MacPaint Document Format
Platforms & Tools

Revised by: Jim Reekes June 1989
Written by: Bill Atkinson 1983

This Technical Note describes the internal format of a MacPaint® document, which is a standard used by many other programs. This description is the same as that found in the "Macintosh Miscellaneous" section of early *Inside Macintosh* versions.
Changes since October 1988: Fixed bugs in the example code.

MacPaint documents are easy to read and write, and they have become a standard interchange format for full–page images on the Macintosh. This Note describes the MacPaint internal document format to help developers generate and interpret files in this format.

MacPaint documents have a file type of "PNTG," and since they use only the data fork, you can ignore the resource fork. The data fork contains a 512–byte header followed by compressed data which represents a single bitmap (576 pixels wide by 720 pixels tall). At a resolution of 72 pixels per inch, this bitmap occupies the full 8 inch by 10 inch printable area of a standard ImageWriter printer page.

Header

The first 512 bytes of the document form a header of the following format:

- 4–byte version number (default = 2)
- 38*8 = 304 bytes of patterns
- 204 unused bytes (reserved for future expansion)

As a Pascal record, the document format could look like the following:

```
MPHeader = RECORD
       Version:      LONGINT;
       PatArray:     ARRAY [1..38] of Pattern;
       Future:       PACKED ARRAY [1..204] of SignedByte;
END;
```

If the version number is zero, the document uses default patterns, so you can ignore the rest of the header block, and if your program generates MacPaint documents, you can write 512 bytes of zero for the document header. Most programs which read MacPaint documents can skip the header when reading.

Bitmap

Following the header are 720 compressed scan lines of data which form the 576 pixel wide by 720 pixel tall bitmap. Without compression, this bitmap would occupy 51,840 bytes and chew up disk space pretty fast; typical MacPaint documents compress to about 10K using the `_PackBits` procedure to compress runs of equal bytes within each scan line. The bitmap part of a MacPaint document is simply the output of `_PackBits` called 720 times, with 72 bytes of input each time.

To determine the maximum size of a MacPaint file, it is worth noting what *Inside Macintosh* says about `_PackBits`:

“The worst case would be when `_PackBits` adds one byte to the row of bytes when packing.”

If we include an extra 512 bytes for the file header information to the size of an uncompressed bitmap (51,840), then the total number of bytes would be 52,352. If we take into account the extra 720 “potential” bytes (one for each row) to the previous total, the maximum size of a MacPaint file becomes 53,072 bytes.

Reading Sample

```
PROCEDURE ReadMPFile;
{ This is a small example procedure written in Pascal that demonstrates
  how to read MacPaint files. As a final step, it takes the data that
  was read and displays it on the screen to show that it worked.
  Caveat: This is not intended to be an example of good programming
  practice, in that the possible errors merely cause the program to exit.
  This is VERY uninformative, and there should be some sort of error handler
  to explain what happened. For simplicity, and thus clarity, those types
  of things were deliberately not included. This example will not work
  on a 128K Macintosh, since memory allocation is done too simplistically.
}

CONST
       DefaultVolume = 0;
       HeaderSize = 512;            { size of MacPaint header in bytes }
       MaxUnPackedSize = 51840;     { maximum MacPaint size in bytes }
                                    { 720 lines * 72 bytes/line }

VAR
       srcPtr:       Ptr;
       dstPtr:       Ptr;
       saveDstPtr:   Ptr;
       lastDestPtr:  Ptr;
       srcFile:      INTEGER;
       srcSize:      LONGINT;
       errCode:      INTEGER;
       scanLine:     INTEGER;
       aPort: GrafPort;
       theBitMap:    BitMap;

BEGIN
       errCode := FSOpen('MP TestFile', DefaultVolume, srcFile); { Open the
                                                                   file. }
       IF errCode <> noErr THEN ExitToShell;
```

```
errcode := SetFPos(srcFile, fsFromStart, HeaderSize);     { Skip the
                                                            header )
IF errCode <> noErr THEN ExitToShell;

errCode := GetEOF(srcFile, srcSize);       { Find out how big the file
                                             is, }
IF errCode <> noErr THEN ExitToShell;      { and figure out source
                                             size. }

srcSize := srcSize - HeaderSize ;          { Remove the header from
                                             count. }
srcPtr := NewPtr(srcSize);                 { Make buffer just the
                                             right size }
IF srcPtr = NIL THEN ExitToShell;

errCode := FSRead(srcFile, srcSize, srcPtr); { Read the data into the
                                               buffer. }
IF errCode <> noErr THEN ExitToShell;      { File marker is past
                                             header. }

errCode := FSClose(srcFile);               { Close the file we just
                                             read. }
IF errCode <> noErr THEN ExitToShell;

{ Create a buffer that will be used for the Destination BitMap. }
dstPtr := NewPtrClear(MaxUnPackedSize);    {MPW library routine, see
                                            TN 219}
IF dstPtr = NIL THEN ExitToShell;
saveDstPtr := dstPtr;

{ Unpack each scan line into the buffer. Note that 720 scan lines are
  guaranteed to be in the file. (They may be blank lines.) In the
  UnPackBits call, the 72 is the count of bytes done when the file was
  created.  MacPaint does one scan line at a time when creating the
  file. The destination pointer is tested each time through the scan
  loop. UnPackBits should increment this pointer by 72, but in the
  case where the packed file is corrupted UnPackBits may end up
  sending bits into uncharted territory.  A temporary pointer
  "lastDstPtr" is used for testing the result.}

FOR scanLine := 1 TO 720 DO BEGIN
     lastDstPtr := dstPtr;
     UnPackBits(srcPtr, dstPtr, 72);       { bumps both pointers }
     IF ORD4(lastDstPtr) + 72 <> ORD4(dstPtr) THEN ExitToShell;
END;

{ The buffer has been fully unpacked. Create a port that we can draw
  into. You should save and restore the current port.  }
OpenPort(@aPort);

{ Create a BitMap out of our saveDstPtr that can be copied to the
  screen. }
theBitMap.baseAddr := saveDstPtr;
theBitMap.rowBytes := 72;           { width of MacPaint picture }
SetPt(theBitMap.bounds.topLeft, 0, 0);
SetPt(theBitMap.bounds.botRight, 72*8, 720); {maximum rectangle}

{ Now use that BitMap and draw the piece of it to the screen.
  Only draw the piece that is full screen size (portRect). }
CopyBits(theBitMap, aPort.portBits, aPort.portRect,
       aPort.portRect, srcCopy, NIL);
```

```
        { We need to dispose of the memory we’ve allocated.  You would not
          dispose of the destPtr if you wish to edit the data.  }
        DisposPtr(srcPtr);    { dispose of the source buffer }
        DisposPtr(dstPtr);    { dispose of the destination buffer }
END;
```

Writing Sample

```
PROCEDURE WriteMPFile;
{ This is a small example procedure written in Pascal that demonstrates how
  to write MacPaint files. It will use the screen as a handy BitMap to be
  written to a file.
}

CONST
        DefaultVolume = 0;
        HeaderSize = 512;     { size of MacPaint header in bytes }
        MaxFileSize = 53072;  { maximum MacPaint file size. }

VAR
        srcPtr:       Ptr;
        dstPtr:       Ptr;
        dstFile:      INTEGER;
        dstSize:      LONGINT;
        errCode:      INTEGER;
        scanLine:     INTEGER;
        aPort: GrafPort;
        dstBuffer:    PACKED ARRAY[1..HeaderSize] OF BYTE;
        I:     LONGINT;
        picturePtr:   Ptr;
        tempPtr:      BigPtr;
        theBitMap:    BitMap;

BEGIN
        { Make an empty buffer that is the picture size. }
        picturePtr := NewPtrClear(MaxFileSize);    {MPW library routine, see
                                                    TN 219}
        IF picturePtr = NIL THEN ExitToShell;

        { Open a port so we can get to the screen's BitMap easily.  You should
          save and restore the current port. }
        OpenPort(@aPort);

        { Create a BitMap out of our dstPtr that can be copied to the screen.
}
        theBitMap.baseAddr := picturePtr;
        theBitMap.rowBytes := 72;           { width of MacPaint picture }
        SetPt(theBitMap.bounds.topLeft, 0, 0);
        SetPt(theBitMap.bounds.botRight, 72*8, 720); {maximum rectangle}

        { Draw the screen over into our picture buffer. }
        CopyBits(aPort.portBits, theBitMap, aPort.portRect,
                aPort.portRect, srcCopy, NIL);

        { Create the file, giving it the right Creator and File type.}
        errCode := Create('MP TestFile', DefaultVolume, 'MPNT', 'PNTG');
        IF errCode <> noErr THEN ExitToShell;

        { Open the data file to be written. }
        errCode := FSOpen(dstFileName, DefaultVolume, dstFile);
        IF errCode <> noErr THEN ExitToShell;
        FOR I := 1 to HeaderSize DO        { Write the header as all zeros. }
             dstBuffer[I] := 0;
        errCode := FSWrite(dstFile, HeaderSize, @dstBuffer);
        IF errCode <> noErr THEN ExitToShell;

        { Now go into a loop where we pack each line of data into the buffer,
          then write that data to the file. We are using the line count of 72
          in order to make the file readable by MacPaint. Note that the
          Pack/UnPackBits can be used for other purposes. }
        srcPtr := theBitMap.baseAddr;             { point at our picture
                                                    BitMap }
        FOR scanLine := 1 to 720 DO
             BEGIN
                dstPtr := @dstBuffer;             { reset the pointer to
                                                    bottom }
                PackBits(srcPtr, dstPtr, 72);            { bumps both ptrs }
                dstSize := ORD(dstPtr)-ORD(@dstBuffer);   { calc packed size }
                errCode := FSWrite(dstFile, dstSize, @dstBuffer);
                IF errCode <> noErr THEN ExitToShell;
             END;

        errCode := FSClose(dstFile);              { Close the file we just
                                                    wrote. }
        IF errCode <> noErr THEN ExitToShell;
END;
```

Further Reference:

- *Inside Macintosh*, Volume I-135, QuickDraw
- *Inside Macintosh*, Volume I-465, Toolbox Utilities
- *Inside Macintosh*, Volume II-77, The File Manager

MacPaint is a registered trademark of Claris Corporation.

The JPEG Still Picture Compression Standard

Gregory K. Wallace
Multimedia Engineering
Digital Equipment Corporation
Maynard, Massachusetts

Submitted in December 1991 for publication in IEEE Transactions on Consumer Electronics

This paper is a revised version of an article by the same title and author which appeared in the April 1991 issue of Communications of the ACM.

Abstract

For the past few years, a joint ISO/CCITT committee known as JPEG (Joint Photographic Experts Group) has been working to establish the first international compression standard for continuous-tone still images, both grayscale and color. JPEG's proposed standard aims to be generic, to support a wide variety of applications for continuous-tone images. To meet the differing needs of many applications, the JPEG standard includes two basic compression methods, each with various modes of operation. A DCT-based method is specified for "lossy'' compression, and a predictive method for "lossless'' compression. JPEG features a simple lossy technique known as the Baseline method, a subset of the other DCT-based modes of operation. The Baseline method has been by far the most widely implemented JPEG method to date, and is sufficient in its own right for a large number of applications. This article provides an overview of the JPEG standard, and focuses in detail on the Baseline method.

1 Introduction

Advances over the past decade in many aspects of digital technology - especially devices for image acquisition, data storage, and bitmapped printing and display - have brought about many applications of digital imaging. However, these applications tend to be specialized due to their relatively high cost. With the possible exception of facsimile, digital images are not commonplace in general-purpose computing systems the way text and geometric graphics are. The majority of modern business and consumer usage of photographs and other types of images takes place through more traditional analog means.

The key obstacle for many applications is the vast amount of data required to represent a digital image directly. A digitized version of a single, color picture at TV resolution contains on the order of one million bytes; 35mm resolution requires ten times that amount. Use of digital images often is not viable due to high storage or transmission costs, even when image capture and display devices are quite affordable.

Modern image compression technology offers a possible solution. State-of-the-art techniques can compress typical images from 1/10 to 1/50 their uncompressed size without visibly affecting image quality. But compression technology alone is not sufficient. For digital image applications involving storage or transmission to become widespread in today's marketplace, a standard image compression method is needed to enable interoperability of equipment from different manufacturers. The CCITT recommendation for today's ubiquitous Group 3 fax machines [17] is a dramatic example of how a standard compression method can enable an important image application. The Group 3 method, however, deals with bilevel images only and does not address photographic image compression.

For the past few years, a standardization effort known by the acronym JPEG, for Joint Photographic Experts Group, has been working toward establishing the first international digital image compression standard for continuous-tone (multilevel) still images, both grayscale and color. The "joint" in JPEG refers to a collaboration between CCITT and ISO. JPEG convenes officially as the ISO committee designated JTC1/SC2/WG10, but operates in close informal collaboration with CCITT SGVIII. JPEG will be both an ISO Standard and a CCITT Recommendation. The text of both will be identical.

Photovideotex, desktop publishing, graphic arts, color facsimile, newspaper wirephoto transmission, medical imaging, and many other continuous-tone image applications require a compression standard in order to develop significantly beyond their present state. JPEG has undertaken the ambitious task of developing a general-purpose compression standard to meet the needs of almost all continuous-tone still-image applications.

If this goal proves attainable, not only will individual applications flourish, but exchange of images across application boundaries will be facilitated. This latter feature will become increasingly important as more image applications are implemented on general-purpose computing systems, which are themselves becoming increasingly interoperable and internetworked. For applications which require specialized VLSI to meet their compression and decompression speed requirements, a common method will provide economies of scale not possible within a single application.

This article gives an overview of JPEG's proposed image-compression standard. Readers without prior knowledge of JPEG or compression based on the Discrete Cosine Transform (DCT) are encouraged to study first the detailed description of the Baseline sequential codec, which is the basis for all of the DCT-based decoders. While this article provides many details, many more are necessarily omitted. The reader should refer to the ISO draft standard [2] before attempting implementation.

Some of the earliest industry attention to the JPEG proposal has been focused on the Baseline sequential codec as a motion image compression method - of the ‘‘intraframe’’ class, where each frame is encoded as a separate image. This class of motion image coding, while providing less compression than ‘‘interframe’’ methods like MPEG, has greater flexibility for video editing. While this paper focuses only on JPEG as a still picture standard (as ISO intended), it is interesting to note that JPEG is likely to become a ‘‘de facto’’ intraframe motion standard as well.

2 Background: Requirements and Selection Process

JPEG's goal has been to develop a method for continuous-tone image compression which meets the following requirements:

1) be at or near the state of the art with regard to compression rate and accompanying image fidelity, over a wide range of image quality ratings, and especially in the range where visual fidelity to the original is characterized as “very good” to “excellent”; also, the encoder should be parameterizable, so that the application (or user) can set the desired compression/quality tradeoff;

2) be applicable to practically any kind of continuous-tone digital source image (i.e. for most practical purposes not be restricted to images of certain dimensions, color spaces, pixel aspect ratios, etc.) and not be limited to classes of imagery with restrictions on scene content, such as complexity, range of colors, or statistical properties;

3) have tractable computational complexity, to make feasible software implementations with viable performance on a range of CPU's, as well as hardware implementations with viable cost for applications requiring high performance;

4) have the following modes of operation:

   - Sequential encoding: each image component is encoded in a single left-to-right, top-to-bottom scan;
   - Progressive encoding: the image is encoded in multiple scans for applications in which transmission time is long, and the viewer prefers to watch the image build up in multiple coarse-to-clear passes;
   - Lossless encoding: the image is encoded to guarantee exact recovery of every source image sample value (even though the result is low compression compared to the lossy modes);
   - Hierarchical encoding: the image is encoded at multiple resolutions so that lower-resolution versions may be accessed without first having to decompress the image at its full resolution.

In June 1987, JPEG conducted a selection process based on a blind assessment of subjective picture quality, and narrowed 12 proposed methods to three. Three informal working groups formed to refine them, and in January 1988, a second, more rigorous selection process [19] revealed that the “ADCT” proposal [11], based on the 8x8 DCT, had produced the best picture quality.

At the time of its selection, the DCT-based method was only partially defined for some of the modes of operation. From 1988 through 1990, JPEG undertook the sizable task of defining, documenting, simulating, testing, validating, and simply agreeing on the plethora of details necessary for genuine interoperability and universality. Further history of the JPEG effort is contained in [6, 7, 9, 18].

3 Architecture of the Proposed Standard

The proposed standard contains the four "modes of operation" identified previously. For each mode, one or more distinct codecs are specified. Codecs within a mode differ according to the precision of source image samples they can handle or the entropy coding method they use. Although the word codec (encoder/decoder) is used frequently in this article, there is no requirement that implementations must include both an encoder and a decoder. Many applications will have systems or devices which require only one or the other.

The four modes of operation and their various codecs have resulted from JPEG's goal of being generic and from the diversity of image formats across applications. The multiple pieces can give the impression of undesirable complexity, but they should actually be regarded as a comprehensive "toolkit" which can span a wide range of continuous-tone image applications. It is unlikely that many implementations will utilize every tool -- indeed, most of the early implementations now on the market (even before final ISO approval) have implemented only the Baseline sequential codec.

The Baseline sequential codec is inherently a rich and sophisticated compression method which will be sufficient for many applications. Getting this minimum JPEG capability implemented properly and interoperably will provide the industry with an important initial capability for exchange of images across vendors and applications.

4 Processing Steps for DCT-Based Coding

Figures 1 and 2 show the key processing steps which are the heart of the DCT-based modes of operation. These figures illustrate the special case of single-component (grayscale) image compression. The reader can grasp the essentials of DCT-based compression by thinking of it as essentially compression of a stream of 8x8 blocks of grayscale image samples. Color image compression can then be approximately regarded as compression of multiple grayscale images, which are either compressed entirely one at a time, or are compressed by alternately interleaving 8x8 sample blocks from each in turn.

For DCT sequential-mode codecs, which include the Baseline sequential codec, the simplified diagrams indicate how single-component compression works in a fairly complete way. Each 8x8 block is input, makes its way through each processing step, and yields output in compressed form into the data stream. For DCT progressive-mode codecs, an image buffer exists prior to the entropy coding step, so that an image can be stored and then parceled out in multiple scans with successively improving quality. For the hierarchical mode of operation, the steps shown are used as building blocks within a larger framework.

4.1 8x8 FDCT and IDCT

At the input to the encoder, source image samples are grouped into 8x8 blocks, shifted from unsigned integers with range $[0, 2^P - 1]$ to signed integers with range $[-2^{P-1}, 2^{P-1}-1]$, and input to the Forward DCT (FDCT). At the output from the decoder, the Inverse DCT (IDCT) outputs 8x8 sample blocks to form the reconstructed image. The following equations are the idealized mathematical definitions of the 8x8 FDCT and 8x8 IDCT:

$$F(u,v) = \frac{1}{4} C(u)C(v)\left[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y) * \cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right] \quad (1)$$

$$f(x,y) = \frac{1}{4}\left[\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v) * \cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right] \quad (2)$$

where: $C(u),\ C(v) = 1/\sqrt{2}$ for $u,v = 0$;

$C(u),\ C(v) = 1$ otherwise.

The DCT is related to the Discrete Fourier Transform (DFT). Some simple intuition for DCT-based compression can be obtained by viewing the FDCT as a harmonic analyzer and the IDCT as a harmonic synthesizer. Each 8x8 block of source image samples is effectively a 64-point discrete signal which is a function of the two spatial dimensions *x* and *y*. The FDCT takes such a signal as its input and decomposes it into 64 orthogonal basis signals. Each contains one of the 64 unique two-dimensional (2D) "spatial frequencies'' which comprise the input signal's "spectrum." The ouput of the FDCT is the set of 64 basis-signal amplitudes or "DCT coefficients" whose values are uniquely determined by the particular 64-point input signal.

The DCT coefficient values can thus be regarded as the relative amount of the 2D spatial frequencies contained in the 64-point input signal. The coefficient with zero frequency in both dimensions is called the "DC coefficient" and the remaining 63 coefficients are called the "AC coefficients.'' Because sample values

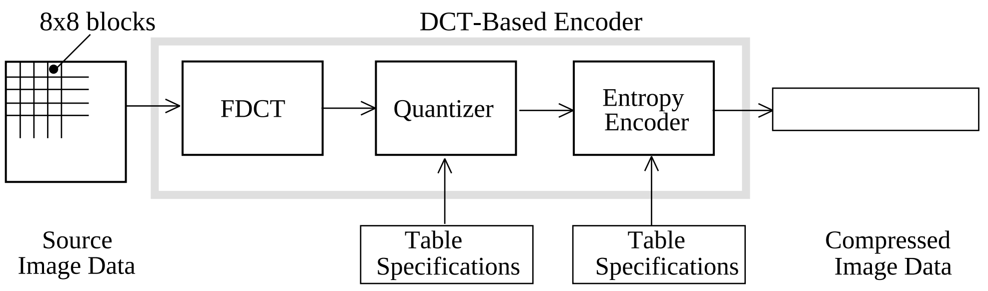


Figure 1. DCT-Based Encoder Processing Steps

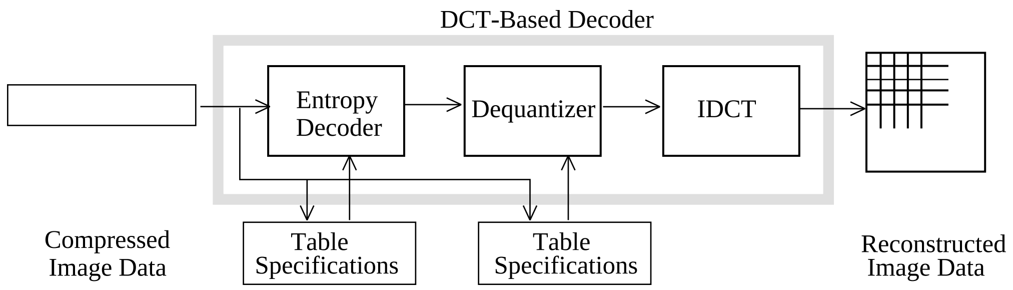


Figure 2. DCT-Based Decoder Processing Steps

typically vary slowly from point to point across an image, the FDCT processing step lays the foundation for achieving data compression by concentrating most of the signal in the lower spatial frequencies. For a typical 8x8 sample block from a typical source image, most of the spatial frequencies have zero or near-zero amplitude and need not be encoded.

At the decoder the IDCT reverses this processing step. It takes the 64 DCT coefficients (which at that point have been quantized) and reconstructs a 64-point ouput image signal by summing the basis signals. Mathematically, the DCT is one-to-one mapping for 64-point vectors between the image and the frequency domains. If the FDCT and IDCT could be computed with perfect accuracy and if the DCT coefficients were not quantized as in the following description, the original 64-point signal could be exactly recovered. In principle, the DCT introduces no loss to the source image samples; it merely transforms them to a domain in which they can be more efficiently encoded.

Some properties of practical FDCT and IDCT implementations raise the issue of what precisely should be required by the JPEG standard. A fundamental property is that the FDCT and IDCT equations contain transcendental functions. Consequently, no physical implementation can compute them with perfect accuracy. Because of the DCT's application importance and its relationship to the DFT, many different algorithms by which the FDCT and IDCT may be approximately computed have been devised [16]. Indeed, research in fast DCT algorithms is ongoing and no single algorithm is optimal for all implementations. What is optimal in software for a general-purpose CPU is unlikely to be optimal in firmware for a programmable DSP and is certain to be suboptimal for dedicated VLSI.

Even in light of the finite precision of the DCT inputs and outputs, independently designed implementations of the very same FDCT or IDCT algorithm which differ even minutely in the precision by which they represent cosine terms or intermediate results, or in the way they sum and round fractional values, will eventually produce slightly different outputs from identical inputs.

To preserve freedom for innovation and customization within implementations, JPEG has chosen to specify neither a unique FDCT algorithm or a unique IDCT algorithm in its proposed standard. This makes compliance somewhat more difficult to confirm, because two compliant encoders (or decoders) generally will not produce identical outputs given identical inputs. The JPEG standard will address this issue by specifying an accuracy test as part of its compliance tests for all DCT-based encoders and decoders; this is to ensure against crudely inaccurate cosine basis functions which would degrade image quality.

For each DCT-based mode of operation, the JPEG proposal specifies separate codecs for images with 8-bit and 12-bit (per component) source image samples. The 12-bit codecs, needed to accommodate certain types of medical and other images, require greater computational resources to achieve the required FDCT or IDCT accuracy. Images with other sample precisions can usually be accommodated by either an 8-bit or 12-bit codec, but this must be done outside the JPEG standard. For example, it would be the responsibility of an application to decide how to fit or pad a 6-bit sample into the 8-bit encoder's input interface, how to unpack it at the decoder's output, and how to encode any necessary related information.

4.2 Quantization

After output from the FDCT, each of the 64 DCT coefficients is uniformly quantized in conjunction with a 64-element Quantization Table, which must be specified by the application (or user) as an input to the encoder. Each element can be any integer value from 1 to 255, which specifies the step size of the quantizer for its corresponding DCT coefficient. The purpose of quantization is to achieve further compression by representing DCT coefficients with no greater precision than is necessary to achieve the desired image quality. Stated another way, the goal of this processing step is to discard information which is not visually significant. Quantization is a many-to-one mapping, and therefore is fundamentally lossy. It is the principal source of lossiness in DCT-based encoders.

Quantization is defined as division of each DCT coefficient by its corresponding quantizer step size, followed by rounding to the nearest integer:

$$F^{Q}(u,v) \;=\; \mathit{Integer\ Round}\,\left(\frac{F(u,v)}{Q(u,v)}\right) \qquad (3)$$

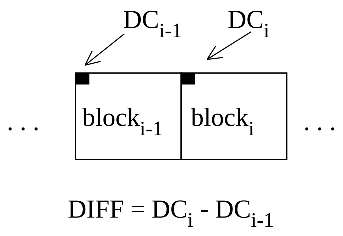
This output value is normalized by the quantizer step size. Dequantization is the inverse function, which in this case means simply that the normalization is removed by multiplying by the step size, which returns the result to a representation appropriate for input to the IDCT:

$$F^{Q'}(u,v) \;=\; F^{Q}(u,v) \;*\; Q(u,v) \qquad (4)$$

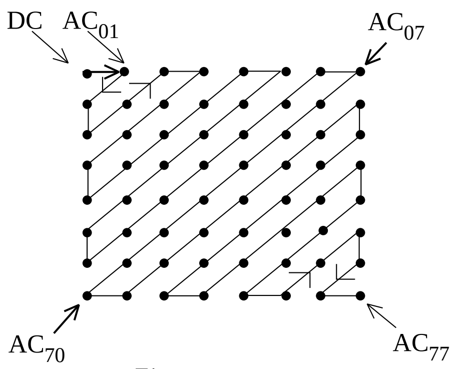
When the aim is to compress the image as much as possible without visible artifacts, each step size ideally should be chosen as the perceptual threshold or "just noticeable difference" for the visual contribution of its corresponding cosine basis function. These thresholds are also functions of the source image characteristics, display characteristics and viewing distance. For applications in which these variables can be reasonably well defined, psychovisual experiments can be performed to determine the best thresholds. The experiment described in [12] has led to a set of Quantization Tables for CCIR-601 [4] images and displays. These have been used experimentally by JPEG members and will appear in the ISO standard as a matter of information, but not as a requirement.

4.3 DC Coding and Zig-Zag Sequence

After quantization, the DC coefficient is treated separately from the 63 AC coefficients. The DC coefficient is a measure of the average value of the 64 image samples. Because there is usually strong correlation between the DC coefficients of adjacent 8x8 blocks, the quantized DC coefficient is encoded as the difference from the DC term of the previous block in the encoding order (defined in the following), as shown in Figure 3. This special treatment is worthwhile, as DC coefficients frequently contain a significant fraction of the total image energy.



Differential DC encoding



Zig−zag sequence

Figure 3. Preparation of Quantized Coefficients for Entropy Coding

Finally, all of the quantized coefficients are ordered into the "zig-zag" sequence, also shown in Figure 3. This ordering helps to facilitate entropy coding by placing low-frequency coefficients (which are more likely to be nonzero) before high-frequency coefficients.

4.4 Entropy Coding

The final DCT-based encoder processing step is entropy coding. This step achieves additional compression losslessly by encoding the quantized DCT coefficients more compactly based on their statistical characteristics. The JPEG proposal specifies two entropy coding methods - Huffman coding [8] and arithmetic coding [15]. The Baseline sequential codec uses Huffman coding, but codecs with both methods are specified for all modes of operation.

It is useful to consider entropy coding as a 2-step process. The first step converts the zig-zag sequence of quantized coefficients into an intermediate sequence of symbols. The second step converts the symbols to a data stream in which the symbols no longer have externally identifiable boundaries. The form and definition of the intermediate symbols is dependent on both the DCT-based mode of operation and the entropy coding method.

Huffman coding requires that one or more sets of Huffman code tables be specified by the application. The same tables used to compress an image are needed to decompress it. Huffman tables may be predefined and used within an application as defaults, or computed specifically for a given image in an initial statistics-gathering pass prior to compression. Such choices are the business of the applications which use JPEG; the JPEG proposal specifies no required Huffman tables. Huffman coding for the Baseline sequential encoder is described in detail in section 7.

By contrast, the particular arithmetic coding method specified in the JPEG proposal [2] requires no tables to be externally input, because it is able to adapt to the image statistics as it encodes the image. (If desired, statistical conditioning tables can be used as inputs for slightly better efficiency, but this is not required.) Arithmetic coding has produced 5-10% better compression than Huffman for many of the images which JPEG members have tested. However, some feel it is more complex than Huffman coding for certain implementations, for example, the highest-speed hardware implementations. (Throughout JPEG's history, "complexity" has proved to be most elusive as a practical metric for comparing compression methods.)

If the only difference between two JPEG codecs is the entropy coding method, transcoding between the two is possible by simply entropy decoding with one method and entropy recoding with the other.

4.5 Compression and Picture Quality

For color images with moderately complex scenes, all DCT-based modes of operation typically produce the following levels of picture quality for the indicated ranges of compression. These levels are only a guideline - quality and compression can vary significantly according to source image characteristics and scene content. (The units "bits/pixel" here mean the total number of bits in the compressed image - including the chrominance components - divided by the number of samples in the luminance component.)

- 0.25-0.5 bits/pixel: moderate to good quality, sufficient for some applications;
- 0.5-0.75 bits/pixel: good to very good quality, sufficient for many applications;
- 0.75-1/5 bits/pixel: excellent quality, sufficient for most applications;
- 1.5-2.0 bits/pixel: usually indistinguishable from the original, sufficient for the most demanding applications.

5 Processing Steps for Predictive Lossless Coding

After its selection of a DCT-based method in 1988, JPEG discovered that a DCT-based lossless mode was difficult to define as a practical standard against which encoders and decoders could be independently implemented, without placing severe constraints on both encoder and decoder implementations.

JPEG, to meet its requirement for a lossless mode of operation, has chosen a simple predictive method which is wholly independent of the DCT processing described previously. Selection of this method was not the result of rigorous competitive evaluation as was the DCT-based method. Nevertheless, the JPEG lossless method produces results which, in light of its simplicity, are surprisingly close to the state of the art for lossless continuous-tone compression, as indicated by a recent technical report [5].

Figure 4 shows the main processing steps for a single-component image. A predictor combines the values of up to three neighboring samples (A, B, and C) to form a prediction of the sample indicated by X in Figure 5. This prediction is then subtracted from the actual value of sample X, and the difference is encoded

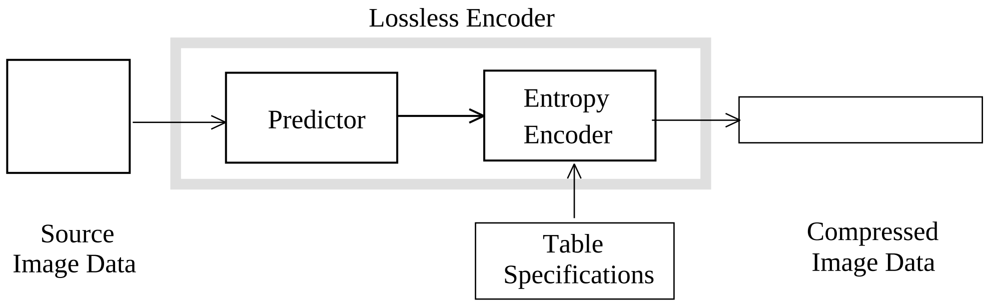


Figure 4. Lossless Mode Encoder Processing Steps

losslessly by either of the entropy coding methods - Huffman or arithmetic. Any one of the eight predictors listed in Table 1 (under "selection-value") can be used.

Selections 1, 2, and 3 are one-dimensional predictors and selections 4, 5, 6 and 7 are two-dimensional predictors. Selection-value 0 can only be used for differential coding in the hierarchical mode of operation. The entropy coding is nearly identical to that used for the DC coefficient as described in section 7.1 (for Huffman coding).

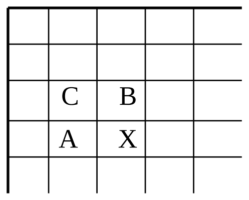


Figure 5. 3-Sample Prediction Neighborhood

For the lossless mode of operation, two different codecs are specified - one for each entropy coding method. The encoders can use any source image precision from 2 to 16 bits/sample, and can use any of the predictors except selection-value 0. The decoders must handle any of the sample precisions and any of the predictors. Lossless codecs typically produce around 2:1 compression for color images with moderately complex scenes.

selection-value	prediction
0	no prediction
1	A
2	B
3	C
4	A+B-C
5	A+((B-C)/2)
6	B+((A-C)/2)
7	(A+B)/2

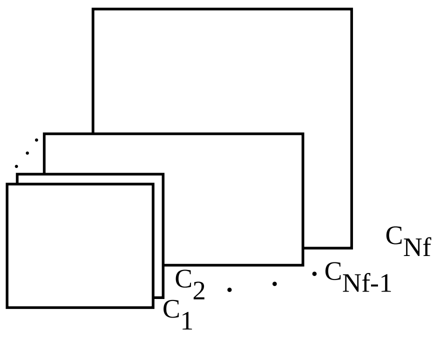
Table 1. Predictors for Lossless Coding

6 Multiple-Component Images

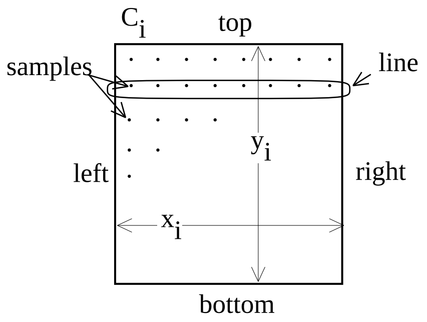
The previous sections discussed the key processing steps of the DCT-based and predictive lossless codecs for the case of single-component source images. These steps accomplish the image data compression. But a good deal of the JPEG proposal is also concerned with the handling and control of color (or other) images with multiple components. JPEG's aim for a generic compression standard requires its proposal to accommodate a variety of source image formats.

6.1 Source Image Formats

The source image model used in the JPEG proposal is an abstraction from a variety of image types and applications and consists of only what is necessary to compress and reconstruct digital image data. The reader should recognize that the JPEG compressed data format does not encode enough information to serve as a complete image representation. For example, JPEG does not specify or encode any information on pixel aspect ratio, color space, or image acquisition characteristics.



(a) Source image with multiple components



(b) Characteristics of an image component

Figure 6. JPEG Source Image Model

Figure 6 illustrates the JPEG source image model. A source image contains from 1 to 255 image components, sometimes called color or spectral bands or channels. Each component consists of a rectangular array of samples. A sample is defined to be an unsigned integer with precision P bits, with any value in the range $[0, 2^P-1]$. All samples of all components within the same source image must have the same precision P. P can be 8 or 12 for DCT-based codecs, and 2 to 16 for predictive codecs.

The ith component has sample dimensions x_i by y_i. To accommodate formats in which some image components are sampled at different rates than others, components can have different dimensions. The dimensions must have a mutual integral relationship defined by H_i and V_i, the relative horizontal and vertical sampling factors, which must be specified for each component. Overall image dimensions X and Y are defined as the maximum x_i and y_i for all components in the image, and can be any number up to 2^{16}. H and V are allowed only the integer values 1 through 4. The encoded parameters are X, Y, and H_is and V_is for each components. The decoder reconstructs the dimensions x_i and y_i for each component, according to the following relationship shown in Equation 5:

$$x_i = \left\lceil X \times \frac{H_i}{H_{max}} \right\rceil \text{ and}$$
$$y_i = \left\lceil Y \times \frac{V_i}{V_{max}} \right\rceil \quad (5)$$

where $\lceil \; \rceil$ is the ceiling function.

6.2 Encoding Order and Interleaving

A practical image compression standard must address how systems will need to handle the data during the process of decompression. Many applications need to pipeline the process of displaying or printing multiple-component images in parallel with the process of decompression. For many systems, this is only feasible if the components are interleaved together within the compressed data stream.

To make the same interleaving machinery applicable to both DCT-based and predictive codecs, the JPEG proposal has defined the concept of “data unit.” A data unit is a sample in predictive codecs and an 8x8 block of samples in DCT-based codecs.

The order in which compressed data units are placed in the compressed data stream is a generalization of raster-scan order. Generally, data units are ordered from left-to-right and top-to-bottom according to the orientation shown in Figure 6. (It is the responsibility of applications to define which edges of a source image are top, bottom, left and right.) If an image component is noninterleaved (i.e., compressed without being interleaved with other components), compressed data units are ordered in a pure raster scan as shown in Figure 7.

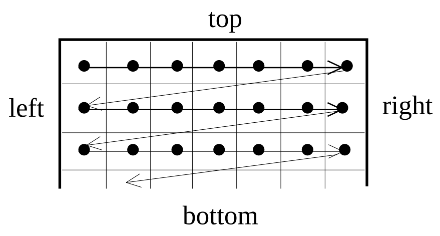
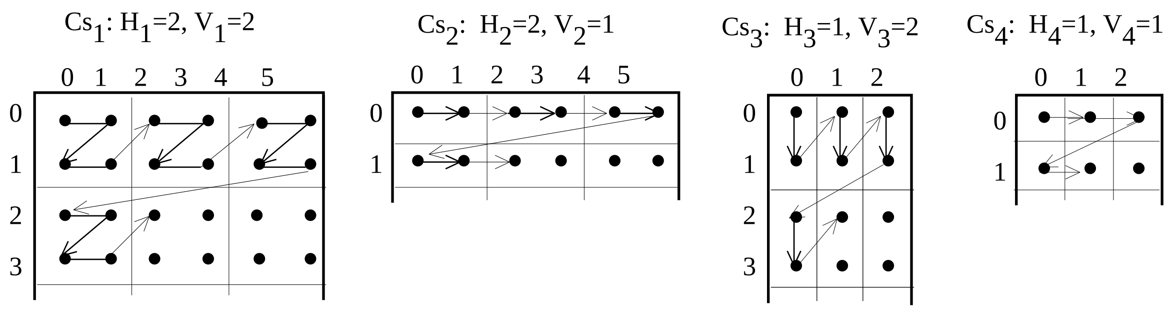


Figure 7. Noninterleaved Data Ordering

When two or more components are interleaved, each component C_i is partitioned into rectangular regions of H_i by V_i data units, as shown in the generalized example of Figure 8. Regions are ordered within a component from left-to-right and top-to-bottom, and within a region, data units are ordered from left-to-right and top-to-bottom. The JPEG proposal defines the term Minimum Coded Unit (MCU) to be the smallest



$$\begin{aligned}
\mathrm{MCU}_1 &= d^1_{00}\, d^1_{01}\, d^1_{10}\, d^1_{11}\quad d^2_{00}\, d^2_{01}\quad d^3_{00}\, d^3_{10}\quad d^4_{00},\\
\mathrm{MCU}_2 &= d^1_{02}\, d^1_{03}\, d^1_{12}\, d^1_{13}\quad d^2_{02}\, d^2_{03}\quad d^3_{01}\, d^3_{11}\quad d^4_{01},\\
\mathrm{MCU}_3 &= d^1_{04}\, d^1_{05}\, d^1_{14}\, d^1_{15}\quad d^2_{04}\, d^2_{05}\quad d^3_{02}\, d^3_{12}\quad d^4_{02},\\
\mathrm{MCU}_4 &= \underbrace{d^1_{20}\, d^1_{21}\, d^1_{30}\, d^1_{31}}_{\mathrm{Cs}_1\text{ data units}}\quad \underbrace{d^2_{10}\, d^2_{11}}_{\mathrm{Cs}_2}\quad \underbrace{d^3_{20}\, d^3_{30}}_{\mathrm{Cs}_3}\quad \underbrace{d^4_{10}}_{\mathrm{Cs}_4},
\end{aligned}$$

Figure 8. Generalized Interleaved Data Ordering Example

group of interleaved data units. For the example shown, MCU_1 consists of data units taken first from the top-left-most region of C_1, followed by data units from the same region of C_2, and likewise for C_3 and C_4. MCU_2 continues the pattern as shown.

Thus, interleaved data is an ordered sequence of MCUs, and the number of data units contained in an MCU is determined by the number of components interleaved and their relative sampling factors. The maximum number of components which can be interleaved is 4 and the maximum number of data units in an MCU is 10. The latter restriction is expressed as shown in Equation 6, where the summation is over the interleaved components:

$$\sum_{\substack{\textit{all}\text{ i in}\\ \text{interleave}}} H_i \times V_i \le 10 \tag{6}$$

Because of this restriction, not every combination of 4 components which can be represented in noninterleaved order within a JPEG-compressed image is allowed to be interleaved. Also, note that the JPEG proposal allows some components to be interleaved and some to be noninterleaved within the same compressed image.

6.3 Multiple Tables

In addition to the interleaving control discussed previously, JPEG codecs must control application of the proper table data to the proper components. The same quantization table and the same entropy coding table (or set of tables) must be used to encode all samples within a component.

JPEG decoders can store up to 4 different quantization tables and up to 4 different (sets of) entropy coding tables simultaneously. (The Baseline sequential decoder is the exception; it can only store up to 2 sets of entropy coding tables.) This is necessary for switching between different tables during decompression of a scan containing multiple (interleaved) components, in order to apply the proper table to the proper component. (Tables cannot be loaded during decompression of a scan.) Figure 9 illustrates the table-switching control that must be managed in conjunction with multiple-component interleaving for the encoder side. (This simplified view does not distinguish between quantization and entropy coding tables.)

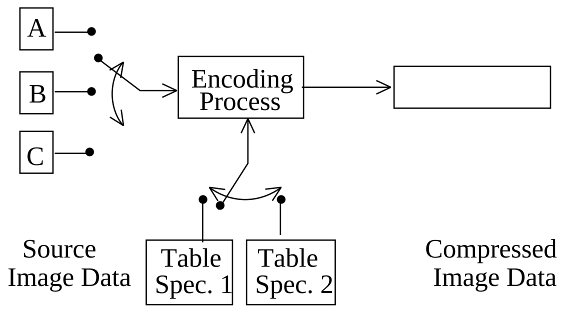


Figure 9. Component-Interleave and Table-Switching Control

7 Baseline and Other DCT Sequential Codecs

The DCT sequential mode of operation consists of the FDCT and Quantization steps from section 4, and the multiple-component control from section 6.3. In addition to the Baseline sequential codec, other DCT sequential codecs are defined to accommodate the two different sample precisions (8 and 12 bits) and the two different types of entropy coding methods (Huffman and arithmetic).

Baseline sequential coding is for images with 8-bit samples and uses Huffman coding only. It also differs from the other sequential DCT codecs in that its decoder can store only two sets of Huffman tables (one AC table and DC table per set). This restriction means that, for images with three or four interleaved components, at least one set of Huffman tables must be shared by two components. This restriction poses no limitation at all for noninterleaved components; a new set of tables can be loaded into the decoder before decompression of a noninterleaved component begins.

For many applications which do need to interleave three color components, this restriction is hardly a limitation at all. Color spaces (YUV, CIELUV, CIELAB, and others) which represent the chromatic (‘‘color’’) information in two components and the achromatic (‘‘grayscale’’) information in a third are more efficient for compression than spaces like RGB. One Huffman table set can be used for the achromatic component and one for the chrominance components. DCT coefficient statistics are similar for the chrominance components of most images, and one set of Huffman tables can encode both almost as optimally as two.

The committee also felt that early availability of single-chip implementations at commodity prices would encourage early acceptance of the JPEG proposal in a variety of applications. In 1988 when Baseline sequential was defined, the committee’s VLSI experts felt that current technology made the feasibility of crowding four sets of loadable Huffman tables - in addition to four sets of Quantization tables - onto a single commodity-priced codec chip a risky proposition.

The FDCT, Quantization, DC differencing, and zig-zag ordering processing steps for the Baseline sequential codec proceed just as described in section 4. Prior to entropy coding, there usually are few nonzero and many zero-valued coefficients. The task of entropy coding is to encode these few coefficients efficiently. The description of Baseline sequential entropy coding is given in two steps: conversion of the quantized DCT coefficients into an intermediate sequence of symbols and assignment of variable-length codes to the symbols.

7.1 Intermediate Entropy Coding Representations

In the intermediate symbol sequence, each nonzero AC coefficient is represented in combination with the ‘‘runlength’’ (consecutive number) of zero-valued AC coefficients which precede it in the zig-zag sequence. Each such runlength/nonzero-coefficient combination is (usually) represented by a pair of symbols:

symbol-1	symbol-2
(RUNLENGTH, SIZE)	(AMPLITUDE)

Symbol-1 represents two pieces of information, RUNLENGTH and SIZE. Symbol-2 represents the single piece of information designated AMPLITUDE, which is simply the amplitude of the nonzero AC coefficient. RUNLENGTH is the number of consecutive zero-valued AC coefficients in the zig-zag sequence preceding the nonzero AC coefficient being represented. SIZE is the number of bits used to encode AMPLITUDE - that is, to encoded symbol-2, by the signed-integer encoding used with JPEG’s particular method of Huffman coding.

RUNLENGTH represents zero-runs of length 0 to 15. Actual zero-runs in the zig-zag sequence can be greater than 15, so the symbol-1 value (15, 0) is interpreted as the extension symbol with runlength=16. There can be up to three consecutive (15, 0) extensions before the terminating symbol-1 whose RUNLENGTH value completes the actual runlength. The terminating symbol-1 is always followed by a single symbol-2, except for the case in which the last run of zeros includes the last (63d) AC coefficient. In this frequent case, the special symbol-1 value (0,0) means EOB (end of block), and can be viewed as an ‘‘escape’’ symbol which terminates the 8x8 sample block.

Thus, for each 8x8 block of samples, the zig-zag sequence of 63 quantized AC coefficients is represented as a sequence of symbol-1, symbol-2 symbol pairs, though each ‘‘pair’’ can have repetitions of symbol-1 in the case of a long run-length or only one symbol-1 in the case of an EOB.

The possible range of quantized AC coefficients determines the range of values which both the AMPLITUDE and the SIZE information must represent. A numerical analysis of the 8x8 FDCT equation shows that, if the 64-point (8x8 block) input signal contains N-bit integers, then the nonfractional part of the output numbers (DCT coefficients) can grow by at most 3 bits. This is also the largest possible size of a quantized DCT coefficient when its quantizer step size has integer value 1.

Baseline sequential has 8-bit integer source samples in the range $[-2^7, 2^7-1]$, so quantized AC coefficient amplitudes are covered by integers in the range $[-2^{10}, 2^{10}-1]$. The signed-integer encoding uses symbol-2 AMPLITUDE codes of 1 to 10 bits in length (so SIZE also represents values from 1 to 10), and RUNLENGTH represents values from 0 to 15 as discussed previously. For AC coefficients, the structure of the symbol-1 and symbol-2 intermediate representations is illustrated in Tables 2 and 3, respectively.

The intermediate representation for an 8x8 sample block’s differential DC coefficient is structured similarly. Symbol-1, however, represents only SIZE information; symbol-2 represents AMPLITUDE information as before:

symbol-1	symbol-2
(SIZE)	(AMPLITUDE)

Because the DC coefficient is differentially encoded, it is covered by twice as many integer values, $[-2^{11}, 2^{11}-1]$ as the AC coefficients, so one additional level must be added to the bottom of Table 3 for DC coefficients. Symbol-1 for DC coefficients thus represents a value from 1 to 11.

		SIZE					
		0	1	2	. . .	9	10
RUN LENGTH	0	EOB					
	.	X	RUN-SIZE values				
	.	X					
	.	X					
	15	ZRL					

Table 2. Baseline Huffman Coding Symbol-1 Structure

7.2 Variable-Length Entropy Coding

Once the quantized coefficient data for an 8x8 block is represented in the intermediate symbol sequence described above, variable-length codes are assigned. For each 8x8 block, the DC coefficient’s symbol-1 and symbol-2 representation is coded and output first.

For both DC and AC coefficients, each symbol-1 is encoded with a variable-length code (VLC) from the Huffman table set assigned to the 8x8 block’s image component. Each symbol-2 is encoded with a “variable-length integer” (VLI) code whose length in bits is given in Table 3. VLCs and VLIs both are codes with variable lengths, but VLIs are not Huffman codes. An important distinction is that the length of a VLC (Huffman code) is not known until it is decoded, but the length of a VLI is stored in its preceding VLC.

Huffman codes (VLCs) must be specified externally as an input to JPEG encoders. (Note that the form in which Huffman tables are represented in the data stream is an indirect specification with which the decoder must construct the tables themselves prior to decompression.) The JPEG proposal includes an example set of Huffman tables in its information annex, but because they are application-specific, it specifies none for required use. The VLI codes in contrast, are “hardwired” into the proposal. This is appropriate, because the VLI codes are far more numerous, can be computed rather than stored, and have not been shown to be appreciably more efficient when implemented as Huffman codes.

7.3 Baseline Encoding Example

This section gives an example of Baseline compression and encoding of a single 8x8 sample block. Note that a good deal of the operation of a complete JPEG Baseline encoder is omitted here, including creation of Interchange Format information (parameters, headers, quantization and Huffman tables), byte-stuffing, padding to byte-boundaries prior to a marker code, and other key operations. Nonetheless, this example should help to make concrete much of the foregoing explanation.

Figure 10(a) is an 8x8 block of 8-bit samples, aribtrarily extracted from a real image. The small variations from sample to sample indicate the predominance of low spatial frequencies. After subtracting 128 from each sample for the required level-shift, the 8x8 block is input to the FDCT, equation (1). Figure 10(b) shows (to one decimal place) the resulting DCT coefficients. Except for a few of the lowest frequency coefficients, the amplitudes are quite small.

139	144	149	153	155	155	155	155
144	151	153	156	159	156	156	156
150	155	160	163	158	156	156	156
159	161	162	160	160	159	159	159
159	160	161	162	162	155	155	155
161	161	161	161	160	157	157	157
162	162	161	163	162	157	157	157
162	162	161	161	163	158	158	158

(a) source image samples

235.6	-1.0	-12.1	-5.2	2.1	-1.7	-2.7	1.3
-22.6	-17.5	-6.2	-3.2	-2.9	-0.1	0.4	-1.2
-10.9	-9.3	-1.6	1.5	0.2	-0.9	-0.6	-0.1
-7.1	-1.9	0.2	1.5	0.9	-0.1	0.0	0.3
-0.6	-0.8	1.5	1.6	-0.1	-0.7	0.6	1.3
1.8	-0.2	1.6	-0.3	-0.8	1.5	1.0	-1.0
-1.3	-0.4	-0.3	-1.5	-0.5	1.7	1.1	-0.8
-2.6	1.6	-3.8	-1.8	1.9	1.2	-0.6	-0.4

(b) forward DCT coefficients

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

(c) quantization table

15	0	-1	0	0	0	0	0
-2	-1	0	0	0	0	0	0
-1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(d) normalized quantized coefficients

240	0	-10	0	0	0	0	0
-24	-12	0	0	0	0	0	0
-14	-13	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(e) denormalized quantized coefficients

144	146	149	152	154	156	156	156
148	150	152	154	156	156	156	156
155	156	157	158	158	157	156	155
160	161	161	162	161	159	157	155
163	163	164	163	162	160	158	156
163	164	164	164	162	160	158	157
160	161	162	162	162	161	159	158
158	159	161	161	162	161	159	158

(f) reconstructed image samples

Figure 10. DCT and Quantization Examples

Figure 10(c) is the example quantization table for luminance (grayscale) components included in the informational annex of the draft JPEG standard part 1 [2]. Figure 10(d) shows the quantized DCT coefficients, normalized by their quantization table entries, as specified by equation (3). At the decoder these numbers are "denormalized" according to equation (4), and input to the IDCT, equation (2). Finally, figure 10(f) shows the reconstructed sample values, remarkably similar to the originals in 10(a).

Of course, the numbers in figure 10(d) must be Huffman-encoded before transmission to the decoder. The first number of the block to be encoded is the DC term, which must be differentially encoded. If the quantized DC term of the previous block is, for example, 12, then the difference is +3. Thus, the intermediate representation is (2)(3), for SIZE=2 and AMPLITUDE=3.

Next, the the quantized AC coefficients are encoded. Following the zig-zag order, the first non-zero coefficient is -2, preceded by a zero-run of 1. This yields an intermediate representation of (1,2)(-2). Next encountered in the zig-zag order are three consecutive non-zeros of amplitude -1. This means each is preceded by a zero-run of length zero, for intermediate symbols (0,1)(-1). The last non-zero coefficient is -1 preceded by two zeros, for (2,1)(-1). Because this is the last non-zero coefficient, the final symbol representing this 8x8 block is EOB, or (0,0).

Thus, the intermediate sequence of symbols for this example 8x8 block is:

(2)(3), (1,2)(-2), (0,1)(-1), (0,1)(-1),
(0,1)(-1), (2,1)(-1), (0,0)

Next the codes themselves must be assigned. For this example, the VLCs (Huffman codes) from the informational annex of [2] will be used. The differential-DC VLC for this example is:

(2)	011

The AC luminance VLCs for this example are:

(0,0)	1010
(0,1)	00
(1,2)	11011
(2,1)	11100

The VLIs specified in [2] are related to the two's complement representation. They are:

(3)	11
(-2)	01
(-1)	0

Thus, the bit-stream for this 8x8 example block is as follows. Note that 31 bits are required to represent 64 coefficients, which achieves compression of just under 0.5 bits/sample:

0111111011010000000001110001010

7.4 Other DCT Sequential Codecs

The structure of the 12-bit DCT sequential codec with Huffman coding is a straightforward extension of the entropy coding method described previously. Quantized DCT coefficients can be 4 bits larger, so the SIZE and AMPLITUDE information extend accordingly. DCT sequential with arithmetic coding is described in detail in [2].

8 DCT Progressive Mode

The DCT progressive mode of operation consists of the same FDCT and Quantization steps (from section 4) that are used by DCT sequential mode. The key difference is that each image component is encoded in multiple scans rather than in a single scan. The first scan(s) encode a rough but recognizable version of the image which can be transmitted quickly in comparison to the total transmission time, and are refined by succeeding scans until reaching a level of picture quality that was established by the quantization tables.

To achieve this requires the addition of an image-sized buffer memory at the output of the quantizer, before the input to entropy encoder. The buffer memory must be of sufficient size to store the image as quantized DCT coefficients, each of which (if stored straightforwardly) is 3 bits larger than the source image samples. After each block of DCT coefficients is quantized, it is stored in the coefficient buffer memory. The buffered coefficients are then partially encoded in each of multiple scans.

There are two complementary methods by which a block of quantized DCT coefficients may be partially encoded. First, only a specified "band" of coefficients from the zig-zag sequence need be encoded within a given scan. This procedure is called "spectral selection," because each band typically contains coefficients which occupy a lower or higher part of the spatial-frequency spectrum for that 8x8 block. Secondly, the coefficients within the current band need not be encoded to their full (quantized) accuracy in a given scan. Upon a coefficient's first encoding, the N most significant bits can be encoded first, where N is specifiable. In subsequent scans, the less significant bits can then be encoded. This procedure is called ''successive approximation.'' Both procedures can be used separately, or mixed in flexible combinations.

SIZE	AMPLITUDE
1	-1,1
2	-3,-2,2,3
3	-7..-4,4..7
4	-15..-8,8..15
5	-31..-16,16..31
6	-63..-32,32..63
7	-127..-64,64..127
8	-255..-128,128..255
9	-511..-256,256..511
10	-1023..-512,512..1023

Table 3. Baseline Entropy Coding Symbol-2 Structure

Some intuition for spectral selection and successive approximation can be obtained from Figure 11. The quantized DCT coefficient information can be viewed as a rectangle for which the axes are the DCT coefficients (in zig-zag order) and their amplitudes. Spectral selection slices the information in one dimension and successive approximation in the other.

9 Hierarchical Mode of Operation

The hierarchical mode provides a "pyramidal" encoding of an image at multiple resolutions, each differing in resolution from its adjacent encoding by a factor of two in either the horizontal or vertical dimension or both. The encoding procedure can be summarized as follows:

1) Filter and down-sample the original image by the desired number of multiples of 2 in each dimension.

2) Encode this reduced-size image using one of the sequential DCT, progressive DCT, or lossless encoders described previously.

3) Decode this reduced-size image and then interpolate and up-sample it by 2 horizontally and/or vertically, using the identical interpolation filter which the receiver must use.

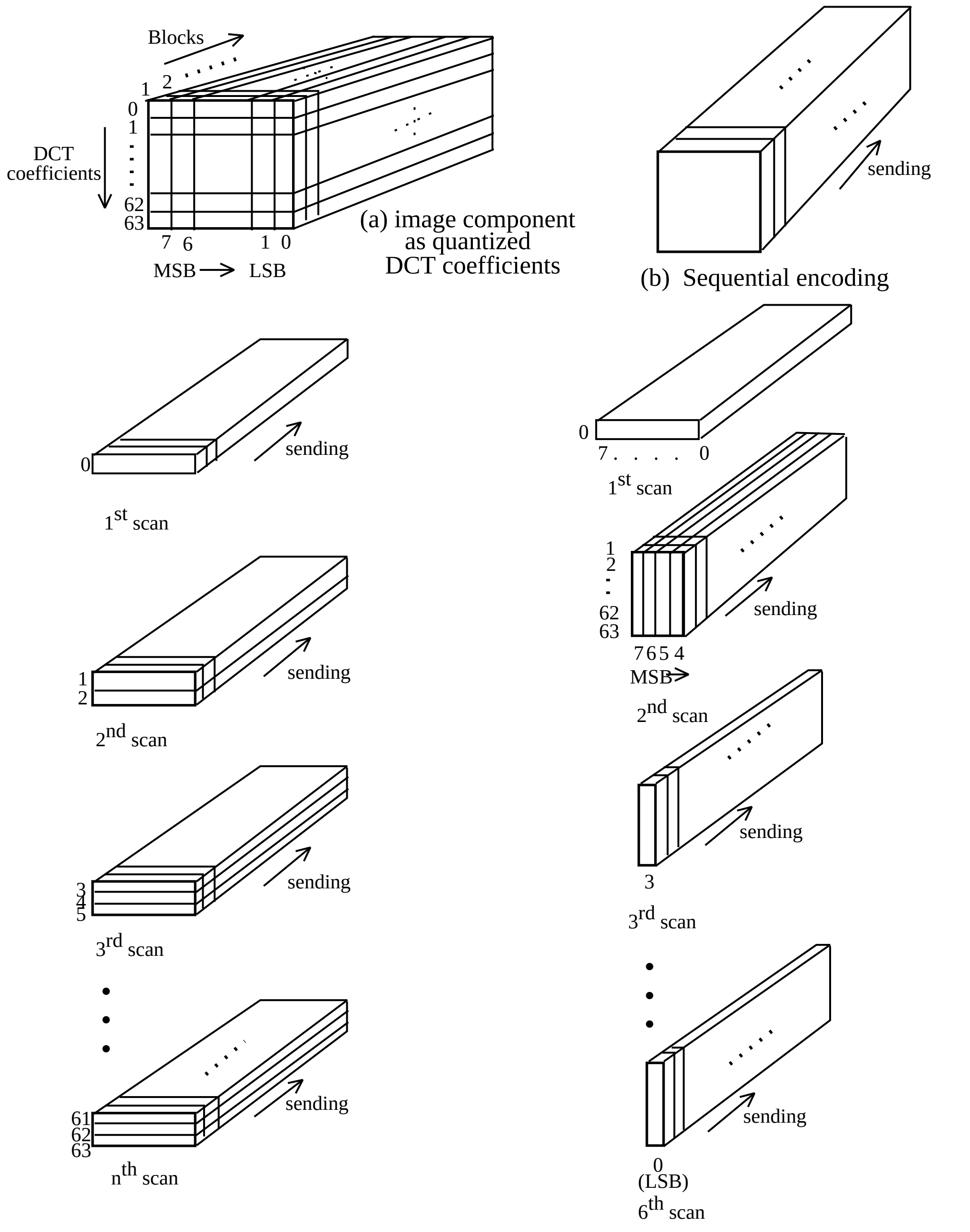

Blocks
DCT coefficients
0
1
62
63
7 6 1 0
MSB LSB
(a) image component as quantized DCT coefficients
sending
(b) Sequential encoding
1st scan
2nd scan
3rd scan
nth scan
61
62
63
7 0
7654
MSB
(LSB)
6th scan
c) progressive encoding: spectral selection
d) progressive encoding: successive approximation

Figure 11. Spectral Selection and Successive Approximation Methods of Progressive Encoding

4) Use this up-sampled image as a prediction of the original at this resolution, and encode the difference image using one of the sequential DCT, progressive DCT, or lossless encoders described previously.

5) Repeat steps 3) and 4) until the full resolution of the image has been encoded.

The encoding in steps 2) and 4) must be done using only DCT-based processes, only lossless processes, or DCT-based processes with a final lossless process for each component.

Hierarchical encoding is useful in applications in which a very high resolution image must be accessed by a lower-resolution display. An example is an image scanned and compressed at high resolution for a very high-quality printer, where the image must also be displayed on a low-resolution PC video screen.

10 Other Aspects of the JPEG Proposal

Some key aspects of the proposed standard can only be mentioned briefly. Foremost among these are points concerning the coded representation for compressed image data specified in addition to the encoding and decoding procedures.

Most importantly, an *interchange format* syntax is specified which ensures that a JPEG-compressed image can be exchanged successfully between different application environments. The format is structured in a consistent way for all modes of operation. The interchange format always includes all quantization and entropy-coding tables which were used to compress the image.

Applications (and application-specific standards) are the "users" of the JPEG standard. The JPEG standard imposes no requirement that, within an application's environment, all or even any tables must be encoded with the compressed image data during storage or transmission. This leaves applications the freedom to specify default or referenced tables if they are considered appropriate. It also leaves them the responsibility to ensure that JPEG-compliant decoders used within their environment get loaded with the proper tables at the proper times, and that the proper tables are included in the interchange format when a compressed image is "exported" outside the application.

Some of the important applications that are already in the process of adopting JPEG compression or have stated their interest in doing so are Adobe's PostScript language for printing systems [1], the Raster Content portion of the ISO Office Document Architecture and Interchange Format [13], the future CCITT color facsimile standard, and the European ETSI videotext standard [10].

11 Standardization Schedule

JPEG's ISO standard will be divided into two parts. Part 1 [2] will specify the four modes of operation, the different codecs specified for those modes, and the interchange format. It will also contain a substantial informational section on implementation guidelines. Part 2 [3] will specify the compliance tests which will determine whether an encoder implementation, a decoder implementation, or a JPEG-compressed image in interchange format comply with the Part 1 specifications. In addition to the ISO documents referenced, the JPEG standard will also be issued as CCITT Recommendation T.81.

There are two key balloting phases in the ISO standardization process: a Committee Draft (CD) is balloted to determine promotion to Draft International Standard (DIS), and a DIS is balloted to determine promotion to International Standard (IS). A CD ballot requires four to six months of processing, and a DIS ballot requires six to nine months of processing. JPEG's Part 1 began DIS ballot in November 1991, and Part 2 began CD ballot in December 1991.

Though there is no guarantee that the first ballot of each phase will result in promotion to the next, JPEG achieved promotion of CD Part 1 to DIS Part 1 in the first ballot. Moreover, JPEG's DIS Part 1 has undergone no technical changes (other than some minor corrections) since JPEG's final Working Draft (WD) [14]. Thus, Part 1 has remained unchanged from the final WD, through CD, and into DIS. If all goes well, Part 1 should receive final approval as an IS in mid-1992, with Part 2 getting final IS approval about nine months later.

12 Conclusions

The emerging JPEG continuous-tone image compression standard is not a panacea that will solve the myriad issues which must be addressed before digital images will be fully integrated within all the applications that will ultimately benefit from them. For example, if two applications cannot exchange uncompressed images because they use incompatible color spaces, aspect ratios, dimensions, etc. then a common compression method will not help.

However, a great many applications are "stuck" because of storage or transmission costs, because of argument over which (nonstandard) compression method to use, or because VLSI codecs are too expensive due to low volumes. For these applications, the thorough technical evaluation, testing, selection, validation, and documentation work which JPEG committee members have performed is expected to soon yield an approved international standard that will withstand the tests of quality and time. As diverse imaging applications become increasingly implemented on open networked computing systems, the ultimate measure of the committee's success will be when JPEG-compressed digital images come to be regarded and even taken for granted as "just another data type," as text and graphics are today.

For more information

Information on how to obtain the ISO JPEG (draft) standards can be obtained by writing the author at the following address:

Digital Equipment Corporation
146 Main Street, ML01-2/U44
Maynard, MA 01754-2571

Internet: wallace@gauss.enet.dec.com

Floppy disks containing uncompressed, compressed, and reconstructed data for the purpose of informally validating whether an encoder or decoder implementation conforms to the proposed standard are available. Thanks to the following JPEG committee member and his company who have agreed to provide these for a nominal fee on behalf of the committee until arrangements can be made for ISO to provide them:

Eric Hamilton
C-Cube Microsystems
1778 McCarthy Blvd.
Milpitas, CA 95035

Acknowledgments

The following longtime JPEG core members have spent untold hours (usually in addition to their ‘‘real jobs’’) to make this collaborative international effort succeed. Each has made specific substantive contributions to the JPEG proposal: Aharon Gill (Zoran, Israel), Eric Hamilton (C-Cube, USA), Alain Leger (CCETT, France), Adriaan Ligtenberg (Storm, USA), Herbert Lohscheller (ANT, Germany), Joan Mitchell (IBM, USA), Michael Nier (Kodak, USA), Takao Omachi (NEC, Japan), William Pennebaker (IBM, USA), Henning Poulsen (KTAS, Denmark), and Jorgen Vaaben (AutoGraph, Denmark). The leadership efforts of Hiroshi Yasuda (NTT, Japan), the Convenor of JTC1/SC2/WG8 from which JPEG was spawned, Istvan Sebestyen (Siemens, Germany), the Special Rapporteur from CCITT SGVIII, and Graham Hudson (British Telecom U.K.) former JPEG chair and founder of the effort which became JPEG. The author regrets that space does not permit recognition of the many other individuals who contributed to JPEG’s work.

Thanks to Majid Rabbani of Eastman Kodak for providing the example in section 7.3.

The author’s role within JPEG has been supported in a great number of ways by Digital Equipment Corporation

References

1. Adobe Systems Inc. *PostScript Language Reference Manual*. Second Ed. Addison Wesley, Menlo Park, Calif. 1990
2. Digital Compression and Coding of Continuous-tone Still Images, Part 1, Requirements and Guidelines. ISO/IEC JTC1 Draft International Standard 10918-1, Nov. 1991.
3. Digital Compression and Coding of Continuous-tone Still Images, Part 2, Compliance Testing. ISO/IEC JTC1 Committee Draft 10918-2, Dec. 1991.
4. Encoding parameters of digital television for studios. CCIR Recommendations, Recommendation 601, 1982.
5. Howard, P.G., and Vitter, J.S. New methods for lossless image compression using arithmetic coding. Brown University Dept. of Computer Science Tech. Report No. CS-91-47, Aug. 1991.
6. Hudson, G.P. The development of photographic videotex in the UK. In *Proceedings of the IEEE Global Telecommunications Conference, IEEE Communication Society,* 1983, pp. 319-322.
7. Hudson, G.P., Yasuda, H., and Sebestyén, I. The international standardization of a still picture compression technique. In *Proceedings of the IEEE Global Telecommunications Conference, IEEE Communications Society,* Nov. 1988, pp. 1016-1021.
8. Huffman, D.A. A method for the construction of minimum redundancy codes. In *Proceedings IRE,* vol. 40, 1962, pp. 1098-1101.
9. Léger, A. Implementations of fast discrete cosine transform for full color videotex services and terminals. In *Proceedings of the IEEE Global Telecommunications Conference, IEEE Communications Society,* 1984, pp. 333-337.
10. Léger, A., Omachi, T., and Wallace, G. The JPEG still picture compression algorithm. In *Optical Engineering*, vol. 30, no. 7 (July 1991), pp. 947-954.
11. Léger, A., Mitchell, M., and Yamazaki, Y. Still picture compression algorithms evaluated for international standardization. In P*roceedings of the IEEE Global Telecommunications Conference, IEEE Communications Society,* Nov. 1988, pp. 1028-1032.
12. Lohscheller, H. A subjectively adapted image communication system. *IEEE Trans. Commun.* COM-32 (Dec. 1984), pp. 1316-1322.

13. Office Document Architecture (ODA) and Interchange Format, Part 7: Raster Graphics Content Architectures. ISO/IEC JTC1 International Standard 8613-7.

14. Pennebaker, W.B., JPEG Tech. Specification, Revision 8. Informal Working paper JPEG-8-R8, Aug. 1990.

15. Pennebaker, W.B., Mitchell, J.L., et. al. Arithmetic coding articles. *IBM J. Res. Dev.*, vol. 32, no. 6 (Nov. 1988), pp. 717-774.

16. Rao, K.R., and Yip, P. *Discrete Cosine Transform--Algorithms, Advantages, Applications*. Academic Press, Inc. London, 1990.

17. Standardization of Group 3 facsimile apparatus for document transmission. CCITT Recommendations, Fascicle VII.2, Recommendation T.4, 1980.

18. Wallace, G.K. Overview of the JPEG (ISO/CCITT) still image compression standard. Image Processing Algorithms and Techniques. In *Proceedings of the SPIE*, vol. 1244 (Feb. 1990), pp. 220-233.

19. Wallace, G., Vivian, R,. and Poulsen, H. Subjective testing results for still picture compression algorithms for international standardization. In *Proceedings of the IEEE Global Telecommunications Conference. IEEE Communications Society*, Nov. 1988, pp. 1022-1027.

Biography

Gregory K. Wallace is currently Manager of Multimedia Engineering, Advanced Development, at Digital Equipment Corporation. Since 1988 he has served as Chair of the JPEG committee (ISO/IEC JTC1/SC2/WG10). For the past five years at DEC, he has worked on efficient software and hardware implementations of image compression and processing algorithms for incorporation in general-purpose computing systems. He received the BSEE and MSEE from Stanford University in 1977 and 1979. His current research interests are the integration of robust real-time multimedia capabilities into networked computing systems.

Welcome to the JPEG Tutorial!

This page presents a brief description of how JPEG compresses images. JPEG, unlike other formats like PPM, PGM, and GIF, is a ***lossy*** compression technique; this means visual information is lost permanently. The key to making JPEG work is choosing what data to throw away.

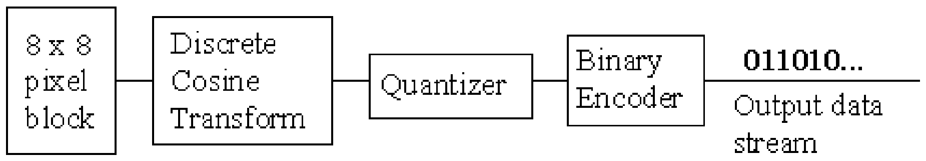
Introduction

JPEG is the image compression standard developed by the Joint Photographic Experts Group. It works best on natural images (scenes). This tutorial describes general JPEG compression for greyscale images; however, JPEG compresses color images just as easily. For instance, It compresses the red-green-blue parts of a color image as three separate greyscale images - each compressed to a different extent, if desired. The section, comparison of JPEG images, displays both color and black and white images compressed with JPEG.

How JPEG works

Figure one describes the JPEG process. JPEG divides up the image into 8 by 8 pixel blocks, and then calculates the discrete cosine transform (DCT) of each block. A quantizer rounds off the DCT coefficients according to the quantization matrix. This step produces the "lossy" nature of JPEG, but allows for large compression ratios. JPEG's compression technique uses a variable length code on these coefficients, and then writes the compressed data stream to an output file (*.jpg). For decompression, JPEG recovers the quantized DCT coefficients from the compressed data stream, takes the inverse transforms and displays the image. Figure 1 shows this process.

FIGURE 1. BLOCK DIAGRAM OF JPEG COMPRESSION



Testing Methods and Results

What does JPEG look like? How far can one compress an image and have it look identical to the original? presentable? merely recognizable? The comparison of JPEG images answers these questions. This section tests color and b/w pictures compressed at different levels. The original images are in PBM format. Version 3.0 of *XV, the imaging system for X-Windows,* implemented the JPEG compression.

References

Many books and articles discuss JPEG in more detail. An anonymous FTP site for more JPEG documentation is: ftp.uu.net/graphics/jpeg/.

Index of Hyperlinks

Below are all the hyperlinks used in this document.

Comments or Problems

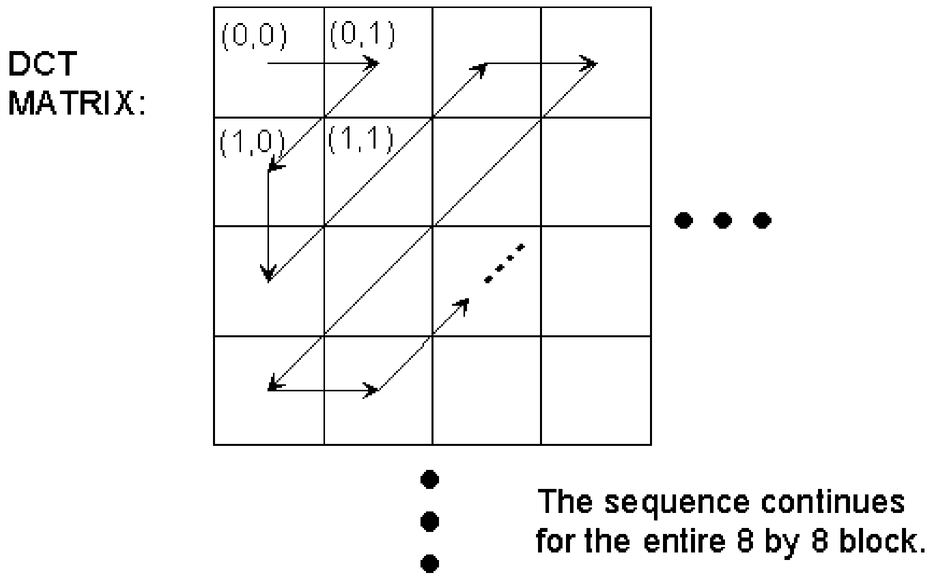
Thank you for reading this JPEG tutorial. If you have any problems with this document, or would like more information, please send email to ace@ecn.purdue.edu.

JPEG’s Compression Technique

Introduction

After quantization, it is not unusual for more than half of the DCT coefficients to equal zero. JPEG incorporates run-length coding to take advantage of this. For each non-zero DCT coefficient, JPEG records the number of zeros that preceded the number, the number of bits needed to represent the number’s amplitude, and the amplitude itself. To consolidate the runs of zeros, JPEG processes DCT coefficients in the zigzag pattern shown in figure two:

FIGURE 2. ZIG-ZAG SEQUENCE FOR BINARY ENCODING



Encoding

The number of previous zeros and the bits needed for the current number’s amplitude form a pair. Each pair has its own code word, assigned through a variable length code (for example Huffman, Shannon-Fano or Arithmetic coding). JPEG outputs the code word of the pair, and then the codeword for the coefficient’s amplitude (also from a variable length code). After each block, JPEG writes a unique end-of-block sequence to the output stream, and moves to the next block. When finished with all blocks, JPEG writes the end-of-file marker.

For Reference:

- The encoder’s Zig-Zag sequence.
- Top level of JPEG Tutorial

The Discrete Cosine Transform

Introduction

The discrete cosine transform (DCT) helps separate the image into parts (or spectral sub-bands) of differing importance (with respect to the image's visual quality). The DCT is similar to the discrete Fourier transform: it transforms a signal or image from the spatial domain to the frequency domain. With an input image, A, the coefficients for the output "image," B, are:

$$B(k_1,k_2) = \sum_{i=0}^{N_1-1} \sum_{j=0}^{N_2-1} 4\cdot A(i,j)\cdot\cos\left[\frac{\pi\cdot k_1}{2\cdot N_1}\cdot(2\cdot i+1)\right]\cdot\cos\left[\frac{\pi\cdot k_2}{2\cdot N_2}\cdot(2\cdot j+1)\right]$$

The input image is N2 pixels wide by N1 pixels high; A(i,j) is the intensity of the pixel in row i and column j; B(k1,k2) is the DCT coefficient in row k1 and column k2 of the DCT matrix. All DCT multiplications are real. This lowers the number of required multiplications, as compared to the discrete Fourier transform. The DCT input is an 8 by 8 array of integers. This array contains each pixel's gray scale level; 8 bit pixels have levels from 0 to 255. The output array of DCT coefficients contains integers; these can range from -1024 to 1023. For most images, much of the signal energy lies at low frequencies; these appear in the upper left corner of the DCT. The lower right values represent higher frequencies, and are often small - small enough to be neglected with little visible distortion.

For Reference:

- Discrete cosine transform equation
- Top level of JPEG Tutorial

GIF

Graphics Interchange Format

Version 89a

Foreword

This document defines the Graphics Interchange Format(sm). The specification given here defines version 89a, which is an extension of version 87a.

The Graphics Interchange Format(sm) as specified here should be considered complete; any deviation from it should be considered invalid, including but not limited to, the use of reserved or undefined fields within control or data blocks, the inclusion of extraneous data within or between blocks, the use of methods or algorithms not specifically listed as part of the format, etc. In general, any and all deviations, extensions or modifications not specified in this document should be considered to be in violation of the format and should be avoided.

Licensing

The Graphics Interchange Format(c) is the copyright property of CompuServe Incorporated. Only CompuServe Incorporated is authorized to define, redefine, enhance, alter, modify or change in any way the definition of the format.

CompuServe Incorporated hereby grants a limited, non-exclusive, royalty-free license for the use of the Graphics Interchange Format(sm) in computer software; computer software utilizing GIF(sm) must acknowledge ownership of the Graphics Interchange Format and its Service Mark by CompuServe Incorporated, in User and Technical Documentation. Computer software utilizing GIF, which is distributed or may be distributed without User or Technical Documentation must display to the screen or printer a message acknowledging ownership of the Graphics Interchange Format and the Service Mark by CompuServe Incorporated; in this case, the acknowledgement may be displayed in an opening screen or leading banner, or a closing screen or trailing banner. A message such as the following may be used:

"The Graphics Interchange Format(c) is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated."

For further information, please contact:

CompuServe Incorporated
Graphics Technology Department
5000 Arlington Center Boulevard
Columbus, Ohio 43220
U. S. A.

CompuServe Incorporated maintains a mailing list with all those individuals and organizations who wish to receive copies of this document when it is corrected or revised. This service is offered free of charge; please provide us with your mailing address.

About this Document

This document describes in detail the definition of the Graphics Interchange Format. This document is intended as a programming reference; it is recommended that the entire document be read carefully before programming, because of the interdependence of the various parts. There is an individual section for each of the Format blocks. Within each section, the sub-section labeled Required Version refers to the version number that an encoder will have to use if the corresponding block is used in the Data Stream. Within each section, a diagram describes the individual fields in the block; the diagrams are drawn vertically; top bytes in the diagram appear first in the Data Stream. Bits within a byte are drawn most significant on the left end. Multi-byte numeric fields are ordered Least Significant Byte first. Numeric constants are represented as Hexadecimal numbers, preceded by "0x". Bit fields within a byte are described in order from most significant bits to least significant bits.

General Description

The Graphics Interchange Format defines a protocol intended for the on-line transmission and interchange of raster graphic data in a way that is independent of the hardware used in their creation or display.

The Graphics Interchange Format is defined in terms of blocks and sub-blocks which contain relevant parameters and data used in the reproduction of a graphic. A GIF Data Stream is a sequence of protocol blocks and sub-blocks representing a collection of graphics. In general, the graphics in a Data Stream are assumed to be related to some degree, and to share some control information; it is recommended that encoders attempt to group together related graphics in order to minimize hardware changes during processing and to minimize control information overhead. For the same reason, unrelated graphics or graphics which require resetting hardware parameters should be encoded separately to the extent possible.

A Data Stream may originate locally, as when read from a file, or it may originate remotely, as when transmitted over a data communications line. The Format is defined with the assumption that an error-free Transport Level Protocol is used for communications; the Format makes no provisions for error-detection and error-correction.

The GIF Data Stream must be interpreted in context, that is, the application program must rely on information external to the Data Stream to invoke the decoder process.

Version Numbers

The version number in the Header of a Data Stream is intended to identify the minimum set of capabilities required of a decoder in order to fully process the Data Stream. An encoder should use the earliest possible version number that includes all the blocks used in the Data Stream. Within each block section in this document, there is an entry labeled Required Version which specifies the earliest version number that includes the corresponding block. The encoder should make every attempt to use the earliest version number covering all the blocks in the Data Stream; the unnecessary use of later version numbers will hinder processing by some decoders.

The Encoder

The Encoder is the program used to create a GIF Data Stream. From raster data and other information, the encoder produces the necessary control and data blocks needed for reproducing the original graphics.

The encoder has the following primary responsibilities.

- Include in the Data Stream all the necessary information to reproduce the graphics.
- Insure that a Data Stream is labeled with the earliest possible Version Number that will cover the definition of all the blocks in it; this is to ensure that the largest number of decoders can process the Data Stream.
- Ensure encoding of the graphics in such a way that the decoding process is optimized. Avoid redundant information as much as possible.
- To the extent possible, avoid grouping graphics which might require resetting hardware parameters during the decoding process.
- Set to zero (off) each of the bits of each and every field designated as reserved. Note that some fields in the Logical Screen Descriptor and the Image Descriptor were reserved under Version 87a, but are used under version 89a.

The Decoder

The Decoder is the program used to process a GIF Data Stream. It processes the Data Stream sequentially, parsing the various blocks and sub-blocks, using the control information to set hardware and process parameters and interpreting the data to render the graphics.

The decoder has the following primary responsibilities.

- Process each graphic in the Data Stream in sequence, without delays other than those specified in the control information.
- Set its hardware parameters to fit, as closely as possible, the control information contained in the Data Stream.

Compliance

An encoder or a decoder is said to comply with a given version of the Graphics Interchange Format if and only if it fully conforms with and correctly implements the definition of the standard associated with that version. An encoder or a decoder may be compliant with a given version number and not compliant with some subsequent version.

About Recommendations

Each block section in this document contains an entry labeled Recommendation; this section lists a set of recommendations intended to guide and organize the use of the particular blocks. Such recommendations are geared towards making the functions of encoders and decoders more efficient, as well as making optimal use of the communications bandwidth. It is advised that these recommendations be followed.

About Color Tables

The GIF format utilizes color tables to render raster-based graphics. A color table can have one of two different scopes: global or local. A Global Color Table is used by all those graphics in the Data Stream which do not have a Local Color Table associated with them. The scope of the Global Color Table is the entire Data Stream. A Local Color Table is always associated with the graphic that immediately follows it; the scope of a Local Color Table is limited to that single graphic. A Local Color Table supersedes a Global Color Table, that is, if a Data Stream contains a Global Color Table, and an image has a Local Color Table associated with it, the decoder must save the Global Color Table, use the Local Color Table to render the image, and then restore the Global Color Table. Both types of color tables are optional, making it possible for a Data Stream to contain numerous graphics without a color table at all. For this reason, it is recommended that the decoder save the last Global Color Table used until another Global Color Table is encountered. In this way, a Data Stream which does not contain either a Global Color Table or a Local Color Table may be processed using the last Global Color Table saved.

If a Global Color Table from a previous Stream is used, that table becomes the Global Color Table of the present Stream. This is intended to reduce the overhead incurred by color tables. In particular, it is recommended that an encoder use only one Global Color Table if all the images in related Data Streams can be rendered with the same table. If no color table is available at all, the decoder is free to use a system color table or a table of its own. In that case, the decoder may use a color table with as many colors as its hardware is able to support; it is recommended that such a table have black and white as its first two entries, so that monochrome images can be rendered adequately.

The Definition of the GIF Format allows for a Data Stream to contain only the Header, the Logical Screen Descriptor, a Global Color Table and the GIF Trailer. Such a Data Stream would be used to load a decoder with a Global Color Table, in preparation for subsequent Data Streams without a color table at all.

Blocks, Extensions and Scope

Blocks can be classified into three groups : Control, Graphic-Rendering and Special Purpose. Control blocks, such as the Header, the Logical Screen Descriptor, the Graphic Control Extension and the Trailer, contain information used to control the process of the Data Stream or information used in setting hardware parameters. Graphic-Rendering blocks such as the Image Descriptor and the Plain Text Extension contain information and data used to render a graphic on the display device. Special Purpose blocks such as the Comment Extension and the Application Extension are neither used to control the process of the Data Stream nor do they contain information or data used to render a graphic on the display device. With the exception of the Logical Screen Descriptor and the Global Color Table, whose scope is the entire Data Stream, all other Control blocks have a limited scope, restricted to the Graphic-Rendering block that follows them. Special Purpose blocks do not delimit the scope of any Control blocks; Special Purpose blocks are transparent to the decoding process. Graphic-Rendering blocks and extensions are used as scope delimiters for Control blocks and extensions. The labels used to identify labeled blocks fall into three ranges : 0x00-0x7F (0-127) are the Graphic Rendering blocks, excluding the Trailer (0x3B); 0x80-0xF9 (128-249) are the Control blocks; 0xFA-0xFF (250-255) are the Special Purpose blocks. These ranges are defined so that decoders can handle block scope by appropriately identifying block labels, even when the block itself cannot be processed.

Block Sizes

The Block Size field in a block, counts the number of bytes remaining in the block, not counting the Block Size field itself, and not counting the Block Terminator, if one is to follow. Blocks other than Data Blocks are intended to be of fixed length; the Block Size field is provided in order to facilitate skipping them, not to allow their size to change in the future. Data blocks and sub-blocks are of variable length to accommodate the amount of data.

Using GIF as an embedded protocol

As an embedded protocol, GIF may be part of larger application protocols, within which GIF is used to render graphics. In such a case, the application protocol could define a block within which the GIF Data Stream would be contained. The application program would then invoke a GIF decoder upon encountering a block of type GIF. This approach is recommended in favor of using Application Extensions, which become overhead for all other applications that do not process them. Because a GIF Data Stream must be processed in context, the application must rely on some means of identifying the GIF Data Stream outside of the Stream itself.

Data Sub-blocks

a. Description. Data Sub-blocks are units containing data. They do not have a label, these blocks are processed in the context of control blocks, wherever data blocks are specified in the format. The first byte of the Data sub-block indicates the number of data bytes to follow. A data sub-block may contain from 0 to 255 data bytes. The size of the block does not account for the size byte itself, therefore, the empty sub-block is one whose size field contains 0x00.

b. Required Version. 87a.

c. Syntax.

```
 7 6 5 4 3 2 1 0 Field Name  Type
    +---------------+
 0  |               | Block Size  Byte
    +---------------+
 1  |               |
    +-             -+
 2  |               |
    +-             -+
 3  |               |
    +-             -+
    |               | Data Values  Byte
    +-             -+
 up |               |
    +-   . . . .   -+
 to |               |
    +-             -+
    |               |
    +-             -+
255 |               |
    +---------------+
```

i) Block Size - Number of bytes in the Data Sub-block; the size must be within 0 and 255 bytes, inclusive.

ii) Data Values - Any 8-bit value. There must be exactly as many Data Values as specified by the Block Size field.

d. Extensions and Scope. This type of block always occurs as part of a larger unit. It does not have a scope of itself.

e. Recommendation. None.

Block Terminator

a. Description. This zero-length Data Sub-block is used to terminate a sequence of Data Sub-blocks. It contains a single byte in the position of the Block Size field and does not contain data.

b. Required Version. 87a.

c. Syntax.

```
 7 6 5 4 3 2 1 0 Field Name  Type
 +---------------+
 0 | | Block Size  Byte
 +---------------+
```

i) Block Size - Number of bytes in the Data Sub-block; this field contains the fixed value 0x00.

ii) Data Values - This block does not contain any data.

d. Extensions and Scope. This block terminates the immediately preceding sequence of Data Sub-blocks. This block cannot be modified by any extension.

e. Recommendation. None.

Header

a. Description. The Header identifies the GIF Data Stream in context. The Signature field marks the beginning of the Data Stream, and the Version field identifies the set of capabilities required of a decoder to fully process the Data Stream. This block is REQUIRED; exactly one Header must be present per Data Stream.

b. Required Version. Not applicable. This block is not subject to a version number. This block must appear at the beginning of every Data Stream.

c. Syntax.

```
 7 6 5 4 3 2 1 0 Field Name  Type
 +---------------+
 0 | | Signature  3 Bytes
 +- -+
 1 | |
 +- -+
 2 | |
 +---------------+
 3 | | Version  3 Bytes
 +- -+
 4 | |
 +- -+
 5 | |
 +---------------+
```

i) Signature - Identifies the GIF Data Stream. This field contains the fixed value 'GIF'.

ii) Version - Version number used to format the data stream. Identifies the minimum set of capabilities necessary to a decoder to fully process the contents of the Data Stream.

iii) Extensions and Scope. The scope of this block is the entire Data Stream. This block cannot be modified by any extension.

d. Recommendations.

i) Signature - This field identifies the beginning of the GIF Data Stream; it is not intended to provide a unique signature for the identification of the data. It is recommended that the GIF Data Stream be identified externally by the application. (Refer to Appendix G for on-line identification of the GIF Data Stream.)

ii) Version - ENCODER : An encoder should use the earliest possible version number that defines all the blocks used in the Data Stream. When two or more Data Streams are combined, the latest of the individual version numbers should be used for the resulting Data Stream. DECODER : A decoder should attempt to process the data stream to the best of its ability; if it encounters a version number which it is not capable of processing fully, it should nevertheless, attempt to process the data stream to the best of its ability, perhaps after warning the user that the data may be incomplete.

Logical Screen Descriptor

a. Description. The Logical Screen Descriptor contains the parameters necessary to define the area of the display device within which the images will be rendered. The coordinates in this block are given with respect to the top-left corner of the virtual screen; they do not necessarily refer to absolute coordinates on the display device. This implies that they could refer to window coordinates in a window-based environment or printer coordinates when a printer is used. This block is REQUIRED; exactly one Logical Screen Descriptor must be present per Data Stream.

b. Required Version. Not applicable. This block is not subject to a version number. This block must appear immediately after the Header.

c. Syntax.

```
 7 6 5 4 3 2 1 0 Field Name  Type
 +---------------+
 0 | | Logical Screen Width Unsigned
 +- -+
 1 | |
 +---------------+
 2 | | Logical Screen Height Unsigned
 +- -+
 3 | |
 +---------------+
 4 | | | | | <Packed Fields> See below
 +---------------+
 5 | | Background Color Index Byte
 +---------------+
 6 | | Pixel Aspect Ratio Byte
 +---------------+

 <Packed Fields> = Global Color Table Flag 1 Bit
  Color Resolution 3 Bits
  Sort Flag  1 Bit
  Size of Global Color Table 3 Bits
```

i) Logical Screen Width - Width, in pixels, of the Logical Screen where the images will be rendered in the displaying device.

ii) Logical Screen Height - Height, in pixels, of the Logical Screen where the images will be rendered in the displaying device.

iii) Global Color Table Flag - Flag indicating the presence of a Global Color Table; if the flag is set, the Global Color Table will immediately follow the Logical Screen Descriptor. This flag also selects the interpretation of the Background Color Index; if the flag is set, the value of the Background Color Index field should be used as the table index of the background color. (This field is the most significant bit of the byte).

Values : 0 - No Global Color Table follows, the Background
Color Index field is meaningless.
1 - A Global Color Table will immediately follow, the
Background Color Index field is meaningful.

iv) Color Resolution - Number of bits per primary color available to the original image, minus 1. This value represents the size of the entire palette from which the colors in the graphic were selected, not the number of colors actually used in the graphic. For example, if the value in this field is 3, then the palette of the original image had 4 bits per primary color available to create the image. This value should be set to indicate the richness of the original palette, even if not every color from the whole palette is available on the source machine.

v) Sort Flag - Indicates whether the Global Color Table is sorted. If the flag is set, the Global Color Table is sorted, in order of decreasing importance. Typically, the order would be decreasing frequency, with most frequent color first. This assists a decoder, with fewer available colors, in choosing the best subset of colors; the decoder may use an initial segment of the table to render the graphic.

Values : 0 - Not ordered.
1 - Ordered by decreasing importance, most
important color first.

vi) Size of Global Color Table - If the Global Color Table Flag is set to 1, the value in this field is used to calculate the number of bytes contained in the Global Color Table. To determine that actual size of the color table, raise 2 to [the value of the field + 1]. Even if there is no Global Color Table specified, set this field according to the above formula so that decoders can choose the best graphics mode to display the stream in. (This field is made up of the 3 least significant bits of the byte.)

vii) Background Color Index - Index into the Global Color Table for the Background Color. The Background Color is the color used for those pixels on the screen that are not covered by an image. If the Global Color Table Flag is set to (zero), this field should be zero and should be ignored.

viii) Pixel Aspect Ratio - Factor used to compute an approximation of the aspect ratio of the pixel in the original image. If the value of the field is not 0, this approximation of the aspect ratio is computed based on the formula:

Aspect Ratio = (Pixel Aspect Ratio + 15) / 64

The Pixel Aspect Ratio is defined to be the quotient of the pixel's width over its height. The value range in this field allows specification of the widest pixel of 4:1 to the tallest pixel of 1:4 in increments of 1/64th.

Values : 0 - No aspect ratio information is given.
1..255 - Value used in the computation.

d. Extensions and Scope. The scope of this block is the entire Data Stream. This block cannot be modified by any extension.

e. Recommendations. None.

Global Color Table

a. Description. This block contains a color table, which is a sequence of bytes representing red-green-blue color triplets. The Global Color Table is used by images without a Local Color Table and by Plain Text Extensions. Its presence is marked by the Global Color Table Flag being set to 1 in the Logical Screen Descriptor; if present, it immediately follows the Logical Screen Descriptor and contains a number of bytes equal to 3 x 2^(Size of Global Color Table+1). This block is OPTIONAL; at most one Global Color Table may be present per Data Stream.

b. Required Version. 87a

c. Syntax.

```
 7 6 5 4 3 2 1 0 Field Name  Type
 +===============+
 0 | | Red 0  Byte
 +- -+
 1 | | Green 0  Byte
 +- -+
 2 | | Blue 0  Byte
 +- -+
 3 | | Red 1  Byte
 +- -+
 | | Green 1  Byte
 +- -+
 up | |
 +- . . . . . -+ ...
 to | |
 +- -+
 | | Green 255  Byte
 +- -+
767 | | Blue 255  Byte
 +===============+
```

d. Extensions and Scope. The scope of this block is the entire Data Stream. This block cannot be modified by any extension.

e. Recommendation. None.

Image Descriptor

a. Description. Each image in the Data Stream is composed of an Image Descriptor, an optional Local Color Table, and the image data. Each image must fit within the boundaries of the Logical Screen, as defined in the Logical Screen Descriptor. The Image Descriptor contains the parameters necessary to process a table based image. The coordinates given in this block refer to coordinates within the Logical Screen, and are given in pixels. This block is a Graphic-Rendering Block, optionally preceded by one or more Control blocks such as the Graphic Control Extension, and may be optionally followed by a Local Color Table; the Image Descriptor is always followed by the image data. This block is REQUIRED for an image. Exactly one Image Descriptor must be present per image in the Data Stream. An unlimited number of images may be present per Data Stream.

b. Required Version. 87a.

c. Syntax.

```
 7 6 5 4 3 2 1 0 Field Name  Type
 +---------------+
 0 | | Image Separator Byte
 +---------------+
 1 | | Image Left Position Unsigned
 +- -+
 2 | |
 +---------------+
 3 | | Image Top Position Unsigned
 +- -+
 4 | |
 +---------------+
 5 | | Image Width  Unsigned
 +- -+
 6 | |
 +---------------+
 7 | | Image Height  Unsigned
 +- -+
 8 | |
 +---------------+
 9 | | | | | | <Packed Fields> See below
 +---------------+

 <Packed Fields> = Local Color Table Flag 1 Bit
```

Interlace Flag 1 Bit
Sort Flag 1 Bit
Reserved 2 Bits
Size of Local Color Table 3 Bits

i) Image Separator - Identifies the beginning of an Image Descriptor. This field contains the fixed value 0x2C.

ii) Image Left Position - Column number, in pixels, of the left edge of the image, with respect to the left edge of the Logical Screen. Leftmost column of the Logical Screen is 0.

iii) Image Top Position - Row number, in pixels, of the top edge of the image with respect to the top edge of the Logical Screen. Top row of the Logical Screen is 0.

iv) Image Width - Width of the image in pixels.

v) Image Height - Height of the image in pixels.

vi) Local Color Table Flag - Indicates the presence of a Local Color Table immediately following this Image Descriptor. (This field is the most significant bit of the byte.)

Values : 0 - Local Color Table is not present. Use
Global Color Table if available.
1 - Local Color Table present, and to follow
immediately after this Image Descriptor.

vii) Interlace Flag - Indicates if the image is interlaced. An image is interlaced in a four-pass interlace pattern; see Appendix E for details.

Values : 0 - Image is not interlaced.
1 - Image is interlaced.

viii) Sort Flag - Indicates whether the Local Color Table is sorted. If the flag is set, the Local Color Table is sorted, in order of decreasing importance. Typically, the order would be decreasing frequency, with most frequent color first. This assists a decoder, with fewer available colors, in choosing the best subset of colors; the decoder may use an initial segment of the table to render the graphic.

Values : 0 - Not ordered.
1 - Ordered by decreasing importance, most
important color first.

ix) Size of Local Color Table - If the Local Color Table Flag is set to 1, the value in this field is used to calculate the number of bytes contained in the Local Color Table. To determine that actual size of the color table, raise 2 to the value of the field + 1. This value should be 0 if there is no Local Color Table specified. (This field is made up of the 3 least significant bits of the byte.)

d. Extensions and Scope. The scope of this block is the Table-based Image Data Block that follows it. This block may be modified by the Graphic Control Extension.

e. Recommendation. None.

Local Color Table

a. Description. This block contains a color table, which is a sequence of bytes representing red-green-blue color triplets. The Local Color Table is used by the image that immediately follows. Its presence is marked by the Local Color Table Flag being set to 1 in the Image Descriptor; if present, the Local Color Table immediately follows the Image Descriptor and contains a number of bytes equal to 3x2^(Size of Local Color Table+1). If present, this color table temporarily becomes the active color table and the following image should be processed using it. This block is OPTIONAL; at most one Local Color Table may be present per Image Descriptor and its scope is the single image associated with the Image Descriptor that precedes it.

b. Required Version. 87a.

c. Syntax.

```
7 6 5 4 3 2 1 0 Field Name  Type
+===============+
0 | | Red 0  Byte
+- -+
1 | | Green 0  Byte
+- -+
2 | | Blue 0  Byte
+- -+
3 | | Red 1  Byte
+- -+
| | Green 1  Byte
+- -+
up | |
+- . . . . -+ ...
```

```
 to | |
 +- -+
 | | Green 255  Byte
 +- -+
767 | | Blue 255  Byte
 +===============+
```

d. Extensions and Scope. The scope of this block is the Table-based Image Data Block that immediately follows it. This block cannot be modified by any extension.

e. Recommendations. None.

Table Based Image Data

a. Description. The image data for a table based image consists of a sequence of sub-blocks, of size at most 255 bytes each, containing an index into the active color table, for each pixel in the image. Pixel indices are in order of left to right and from top to bottom. Each index must be within the range of the size of the active color table, starting at 0. The sequence of indices is encoded using the LZW Algorithm with variable-length code, as described in Appendix F

b. Required Version. 87a.

c. Syntax. The image data format is as follows:

```
 7 6 5 4 3 2 1 0 Field Name  Type
 +---------------+
 | | LZW Minimum Code Size Byte
 +---------------+

 +===============+
 | |
 / / Image Data  Data Sub-blocks
 | |
 +===============+
```

i) LZW Minimum Code Size. This byte determines the initial number of bits used for LZW codes in the image data, as described in Appendix F.

d. Extensions and Scope. This block has no scope, it contains raster data. Extensions intended to modify a Table-based image must appear before the corresponding Image Descriptor.

e. Recommendations. None.

Graphic Control Extension

a. Description. The Graphic Control Extension contains parameters used when processing a graphic rendering block. The scope of this extension is the first graphic rendering block to follow. The extension contains only one data sub-block. This block is OPTIONAL; at most one Graphic Control Extension may precede a graphic rendering block. This is the only limit to the number of Graphic Control Extensions that may be contained in a Data Stream.

b. Required Version. 89a.

c. Syntax.

```
 7 6 5 4 3 2 1 0 Field Name  Type
 +---------------+
0 | | Extension Introducer Byte
 +---------------+
1 | | Graphic Control Label Byte
 +---------------+

 +---------------+
0 | | Block Size  Byte
 +---------------+
1 | | | | | <Packed Fields> See below
 +---------------+
2 | | Delay Time  Unsigned
+- -+
3 | |
 +---------------+
4 | | Transparent Color Index Byte
 +---------------+

 +---------------+
0 | | Block Terminator Byte
 +---------------+
```

```
<Packed Fields> = Reserved  3 Bits
Disposal Method 3 Bits
User Input Flag 1 Bit
Transparent Color Flag 1 Bit
```

i) Extension Introducer - Identifies the beginning of an extension block. This field contains the fixed value 0x21.

ii) Graphic Control Label - Identifies the current block as a Graphic Control Extension. This field contains the fixed value 0xF9.

iii) Block Size - Number of bytes in the block, after the Block Size field and up to but not including the Block Terminator. This field contains the fixed value 4.

iv) Disposal Method - Indicates the way in which the graphic is to be treated after being displayed.

Values : 0 - No disposal specified. The decoder is not required to take any action.

1 - Do not dispose. The graphic is to be left in place.

2 - Restore to background color. The area used by the graphic must be restored to the background color.

3 - Restore to previous. The decoder is required to restore the area overwritten by the graphic with what was there prior to rendering the graphic.

4-7 - To be defined.

v) User Input Flag - Indicates whether or not user input is expected before continuing. If the flag is set, processing will continue when user input is entered. The nature of the User input is determined by the application (Carriage Return, Mouse Button Click, etc.).

Values : 0 - User input is not expected.

1 - User input is expected.

When a Delay Time is used and the User Input Flag is set, processing will continue when user input is received or when the delay time expires, whichever occurs first.

vi) Transparency Flag - Indicates whether a transparency index is given in the Transparent Index field. (This field is the least significant bit of the byte.)

Values : 0 - Transparent Index is not given.

1 - Transparent Index is given.

vii) Delay Time - If not 0, this field specifies the number of hundredths (1/100) of a second to wait before continuing with the processing of the Data Stream. The clock starts ticking immediately after the graphic is rendered. This field may be used in conjunction with the User Input Flag field.

viii) Transparency Index - The Transparency Index is such that when encountered, the corresponding pixel of the display device is not modified and processing goes on to the next pixel. The index is present if and only if the Transparency Flag is set to 1.

ix) Block Terminator - This zero-length data block marks the end of the Graphic Control Extension.

d. Extensions and Scope. The scope of this Extension is the graphic rendering block that follows it; it is possible for other extensions to be present between this block and its target. This block can modify the Image Descriptor Block and the Plain Text Extension.

e. Recommendations.

i) Disposal Method - The mode Restore To Previous is intended to be used in small sections of the graphic; the use of this mode imposes severe demands on the decoder to store the section of the graphic that needs to be saved. For this reason, this mode should be used sparingly. This mode is not intended to save an entire graphic or large areas of a graphic; when this is the case, the encoder should make every attempt to make the sections of the graphic to be restored be separate graphics in the data stream. In the case where a decoder is not capable of saving an area of a graphic marked as Restore To Previous, it is recommended that a decoder restore to the background color.

ii) User Input Flag - When the flag is set, indicating that user input is expected, the decoder may sound the bell (0x07) to alert the user that input is being expected. In the absence of a specified Delay Time, the decoder should wait for user input indefinitely. It is recommended that the encoder not set the User Input Flag without a Delay Time specified.

Comment Extension

a. Description. The Comment Extension contains textual information which is not part of the actual graphics in the GIF Data Stream. It is suitable for including comments about the graphics, credits, descriptions or any other type of non-control and non-graphic data. The Comment Extension may be ignored by the decoder, or it may be saved for later processing; under no circumstances should a Comment Extension disrupt or interfere with the processing of the Data Stream. This block is OPTIONAL; any number of them may appear in the Data Stream.

b. Required Version. 89a.

c. Syntax.

```
 7 6 5 4 3 2 1 0 Field Name  Type
 +---------------+
 0 | | Extension Introducer Byte
 +---------------+
 1 | | Comment Label  Byte
 +---------------+

+===============+
| |
N | | Comment Data  Data Sub-blocks
| |
+===============+

 +---------------+
 0 | | Block Terminator Byte
 +---------------+
```

i) Extension Introducer - Identifies the beginning of an extension block. This field contains the fixed value 0x21.

ii) Comment Label - Identifies the block as a Comment Extension. This field contains the fixed value 0xFE.

iii) Comment Data - Sequence of sub-blocks, each of size at most 255 bytes and at least 1 byte, with the size in a byte preceding the data. The end of the sequence is marked by the Block Terminator.

iv) Block Terminator - This zero-length data block marks the end of the Comment Extension.

d. Extensions and Scope. This block does not have scope. This block cannot be modified by any extension.

e. Recommendations.

i) Data - This block is intended for humans. It should contain text using the 7-bit ASCII character set. This block should not be used to store control information for custom processing.

ii) Position - This block may appear at any point in the Data Stream at which a block can begin; however, it is recommended that Comment Extensions do not interfere with Control or Data blocks; they should be located at the beginning or at the end of the Data Stream to the extent possible.

Plain Text Extension

a. Description. The Plain Text Extension contains textual data and the parameters necessary to render that data as a graphic, in a simple form. The textual data will be encoded with the 7-bit printable ASCII characters. Text data are rendered using a grid of character cells defined by the parameters in the block fields. Each character is rendered in an individual cell. The textual data in this block is to be rendered as mono-spaced characters, one character per cell, with a best fitting font and size. For further information, see the section on Recommendations below. The data characters are taken sequentially from the data portion of the block and rendered within a cell, starting with the upper left cell in the grid and proceeding from left to right and from top to bottom. Text data is rendered until the end of data is reached or the character grid is filled. The Character Grid contains an integral number of cells; in the case that the cell dimensions do not allow for an integral number, fractional cells must be discarded; an encoder must be careful to specify the grid dimensions accurately so that this does not happen. This block requires a Global Color Table to be available; the colors used by this block reference the Global Color Table in the Stream if there is one, or the Global Color Table from a previous Stream, if one was saved. This block is a graphic rendering block, therefore it may be modified by a Graphic Control Extension. This block is OPTIONAL; any number of them may appear in the Data Stream.

b. Required Version. 89a.

c. Syntax.

```
 7 6 5 4 3 2 1 0 Field Name  Type
 +---------------+
0 | | Extension Introducer Byte
 +---------------+
1 | | Plain Text Label Byte
 +---------------+

 +---------------+
0 | | Block Size  Byte
 +---------------+
1 | | Text Grid Left Position Unsigned
+- -+
2 | |
 +---------------+
3 | | Text Grid Top Position Unsigned
```

```
+- -+
4 | |
+---------------+
5 | | Text Grid Width Unsigned
+- -+
6 | |
+---------------+
7 | | Text Grid Height Unsigned
+- -+
8 | |
+---------------+
9 | | Character Cell Width Byte
+---------------+
10 | | Character Cell Height Byte
+---------------+
11 | | Text Foreground Color Index Byte
+---------------+
12 | | Text Background Color Index Byte
+---------------+

+===============+
| |
N | | Plain Text Data Data Sub-blocks
| |
+===============+

+---------------+
0 | | Block Terminator Byte
+---------------+
```

i) Extension Introducer - Identifies the beginning of an extension block. This field contains the fixed value 0x21.

ii) Plain Text Label - Identifies the current block as a Plain Text Extension. This field contains the fixed value 0x01.

iii) Block Size - Number of bytes in the extension, after the Block Size field and up to but not including the beginning of the data portion. This field contains the fixed value 12.

iv) Text Grid Left Position - Column number, in pixels, of the left edge of the text grid, with respect to the left edge of the Logical Screen.

v) Text Grid Top Position - Row number, in pixels, of the top edge of the text grid, with respect to the top edge of the Logical Screen.

vi) Image Grid Width - Width of the text grid in pixels.

vii) Image Grid Height - Height of the text grid in pixels.
viii) Character Cell Width - Width, in pixels, of each cell in the grid.
ix) Character Cell Height - Height, in pixels, of each cell in the grid.
x) Text Foreground Color Index - Index into the Global Color Table to be used to render the text foreground.
xi) Text Background Color Index - Index into the Global Color Table to be used to render the text background.
xii) Plain Text Data - Sequence of sub-blocks, each of size at most 255 bytes and at least 1 byte, with the size in a byte preceding the data. The end of the sequence is marked by the Block Terminator.
xiii) Block Terminator - This zero-length data block marks the end of the Plain Text Data Blocks.

d. Extensions and Scope. The scope of this block is the Plain Text Data Block contained in it. This block may be modified by the Graphic Control Extension.

e. Recommendations. The data in the Plain Text Extension is assumed to be preformatted. The selection of font and size is left to the discretion of the decoder. If characters less than 0x20 or greater than 0xf7 are encountered, it is recommended that the decoder display a Space character (0x20). The encoder should use grid and cell dimensions such that an integral number of cells fit in the grid both horizontally as well as vertically. For broadest compatibility, character cell dimensions should be around 8x8 or 8x16 (width x height); consider an image for unusual sized text.

Application Extension

a. Description. The Application Extension contains application-specific information; it conforms with the extension block syntax, as described below, and its block label is 0xFF.

b. Required Version. 89a.

c. Syntax.

```
7 6 5 4 3 2 1 0 Field Name  Type
+---------------+
0 | | Extension Introducer Byte
+---------------+
1 | | Extension Label Byte
```

```
 +---------------+

 +---------------+
0 | | Block Size  Byte
 +---------------+
1 | |
+- -+
2 | |
+- -+
3 | | Application Identifier 8 Bytes
+- -+
4 | |
+- -+
5 | |
+- -+
6 | |
+- -+
7 | |
+- -+
8 | |
 +---------------+
9 | |
+- -+
10 | | Appl. Authentication Code 3 Bytes
+- -+
11 | |
 +---------------+

+===============+
| |
| | Application Data Data Sub-blocks
| |
| |
+===============+

 +---------------+
0 | | Block Terminator Byte
 +---------------+
```

i) Extension Introducer - Defines this block as an extension. This field contains the fixed value 0x21.

ii) Application Extension Label - Identifies the block as an Application Extension. This field contains the fixed value 0xFF.

iii) Block Size - Number of bytes in this extension block, following the Block Size field, up to but not including the beginning of the Application Data. This field contains the fixed value 11.

iv) Application Identifier - Sequence of eight printable ASCII characters used to identify the application owning the Application Extension.

v) Application Authentication Code - Sequence of three bytes used to authenticate the Application Identifier. An Application program may use an algorithm to compute a binary code that uniquely identifies it as the application owning the Application Extension.

d. Extensions and Scope. This block does not have scope. This block cannot be modified by any extension.

e. Recommendation. None.

Trailer

a. Description. This block is a single-field block indicating the end of the GIF Data Stream. It contains the fixed value 0x3B.

b. Required Version. 87a.

c. Syntax.

```
 7 6 5 4 3 2 1 0 Field Name  Type
 +---------------+
 0 | | GIF Trailer  Byte
 +---------------+
```

d. Extensions and Scope. This block does not have scope, it terminates the GIF Data Stream. This block may not be modified by any extension.

e. Recommendations. None.

Appendix A: Quick Reference Table

Block Name Required Label Ext. Vers.
Application Extension Opt. (*) 0xFF (255) yes 89a
Comment Extension Opt. (*) 0xFE (254) yes 89a
Global Color Table Opt. (1) none no 87a
Graphic Control Extension Opt. (*) 0xF9 (249) yes 89a
Header Req. (1) none no N/A
Image Descriptor Opt. (*) 0x2C (044) no 87a (89a)
Local Color Table Opt. (*) none no 87a
Logical Screen Descriptor Req. (1) none no 87a (89a)
Plain Text Extension Opt. (*) 0x01 (001) yes 89a
Trailer Req. (1) 0x3B (059) no 87a

Unlabeled Blocks
Header Req. (1) none no N/A
Logical Screen Descriptor Req. (1) none no 87a (89a)
Global Color Table Opt. (1) none no 87a
Local Color Table Opt. (*) none no 87a

Graphic-Rendering Blocks
Plain Text Extension Opt. (*) 0x01 (001) yes 89a
Image Descriptor Opt. (*) 0x2C (044) no 87a (89a)

Control Blocks
Graphic Control Extension Opt. (*) 0xF9 (249) yes 89a

Special Purpose Blocks
Trailer Req. (1) 0x3B (059) no 87a
Comment Extension Opt. (*) 0xFE (254) yes 89a
Application Extension Opt. (*) 0xFF (255) yes 89a

legend: (1) if present, at most one occurrence
  (*) zero or more occurrences
  (+) one or more occurrences

Notes : The Header is not subject to Version Numbers.
(89a) The Logical Screen Descriptor and the Image Descriptor retained their syntax from version 87a to version 89a, but some fields reserved under version 87a are used under version 89a.

Appendix B: GIF Grammar

A Grammar is a form of notation to represent the sequence in which certain objects form larger objects. A grammar is also used to represent the number of objects that can occur at a given position. The grammar given here represents the sequence of blocks that form the GIF Data Stream. A grammar is given by listing its rules. Each rule consists of the left-hand side, followed by some form of equals sign, followed by the right-hand side. In a rule, the right-hand side describes how the left-hand side is defined. The right-hand side consists of a sequence of entities, with the possible presence of special symbols. The following legend defines the symbols used in this grammar for GIF.

Legend: <> grammar word
  ::= defines symbol
  * zero or more occurrences
  + one or more occurrences
  | alternate element
  [] optional element

Example:

<GIF Data Stream> ::= Header <Logical Screen> <Data>* Trailer

This rule defines the entity <GIF Data Stream> as follows. It must begin with a Header. The Header is followed by an entity called Logical Screen, which is defined below by another rule. The Logical Screen is followed by the entity Data, which is also defined below by another rule. Finally, the entity Data is followed by the Trailer. Since there is no rule defining the Header or the Trailer, this means that these blocks are defined in the document. The entity Data has a special symbol (*) following it which means that, at this position, the entity Data may be repeated any number of times, including 0 times. For further reading on this subject, refer to a standard text on Programming Languages.

The Grammar

<GIF Data Stream> ::= Header <Logical Screen> <Data>* Trailer

<Logical Screen> ::= Logical Screen Descriptor [Global Color Table]

<Data> ::= <Graphic Block> |
  <Special-Purpose Block>

<Graphic Block> ::= [Graphic Control Extension] <Graphic-Rendering Block>

<Graphic-Rendering Block> ::= <Table-Based Image> |
  Plain Text Extension

<Table-Based Image> ::= Image Descriptor [Local Color Table] Image Data

<Special-Purpose Block> ::= Application Extension |
  Comment Extension

NOTE : The grammar indicates that it is possible for a GIF Data Stream to contain the Header, the Logical Screen Descriptor, a Global Color Table and the GIF Trailer. This special case is used to load a GIF decoder with a Global Color Table, in preparation for subsequent Data Streams without color tables at all.

Appendix C: Glossary

Active Color Table - Color table used to render the next graphic. If the next graphic is an image which has a Local Color Table associated with it, the active color table becomes the Local Color Table associated with that image. If the next graphic is an image without a Local Color Table, or a Plain Text Extension, the active color table is the Global Color Table associated with the Data Stream, if there is one; if there is no Global Color Table in the Data Stream, the active color table is a color table saved from a previous Data Stream, or one supplied by the decoder.

Block - Collection of bytes forming a protocol unit. In general, the term includes labeled and unlabeled blocks, as well as Extensions.

Data Stream - The GIF Data Stream is composed of blocks and sub-blocks representing images and graphics, together with control information to render them on a display device. All control and data blocks in the Data Stream must follow the Header and must precede the Trailer.

Decoder - A program capable of processing a GIF Data Stream to render the images and graphics contained in it.

Encoder - A program capable of capturing and formatting image and graphic raster data, following the definitions of the Graphics Interchange Format.

Extension - A protocol block labeled by the Extension Introducer 0x21.

Extension Introducer - Label (0x21) defining an Extension.

Graphic - Data which can be rendered on the screen by virtue of some algorithm. The term graphic is more general than the term image; in addition to images, the term graphic also includes data such as text, which is rendered using character bit-maps.

Image - Data representing a picture or a drawing; an image is represented by an array of pixels called the raster of the image.

Raster - Array of pixel values representing an image.

Appendix D: Conventions

Animation - The Graphics Interchange Format is not intended as a platform for animation, even though it can be done in a limited way.

Byte Ordering - Unless otherwise stated, multi-byte numeric fields are ordered with the Least Significant Byte first.

Color Indices - Color indices always refer to the active color table, either the Global Color Table or the Local Color Table.

Color Order - Unless otherwise stated, all triple-component RGB color values are specified in Red-Green-Blue order.

Color Tables - Both color tables, the Global and the Local, are optional; if present, the Global Color Table is to be used with every image in the Data Stream for which a Local Color Table is not given; if present, a Local Color Table overrides the Global Color Table. However, if neither color table is present, the application program is free to use an arbitrary color table. If the graphics in several Data Streams are related and all use the same color table, an encoder could place the color table as the Global Color Table in the first Data Stream and leave subsequent Data Streams without a Global Color Table or any Local Color Tables; in this way, the overhead for the table is eliminated. It is recommended that the decoder save the previous Global Color Table to be used with the Data Stream that follows, in case it does not contain either a Global Color Table or any Local Color Tables. In general, this allows the application program to use past color tables, significantly reducing transmission overhead.

Extension Blocks - Extensions are defined using the Extension Introducer code to mark the beginning of the block, followed by a block label, identifying the type of extension. Extension Codes are numbers in the range from 0x00 to 0xFF, inclusive. Special purpose extensions are transparent to the decoder and may be omitted when transmitting the Data Stream on-line. The GIF capabilities dialogue makes the provision for the receiver to request the transmission of all blocks; the default state in this regard is no transmission of Special purpose blocks.

Reserved Fields - All Reserved Fields are expected to have each bit set to zero (off).

Appendix E: Interlaced Images

The rows of an Interlaced images are arranged in the following order:

Group 1 : Every 8th. row, starting with row 0. (Pass 1)
Group 2 : Every 8th. row, starting with row 4. (Pass 2)
Group 3 : Every 4th. row, starting with row 2. (Pass 3)
Group 4 : Every 2nd. row, starting with row 1. (Pass 4)

The Following example illustrates how the rows of an interlaced image are ordered.

Row Number	Interlace Pass
0	1
1	4
2	3
3	4
4	2
5	4
6	3
7	4
8	1
9	4
10	3
11	4
12	2
13	4
14	3
15	4
16	1
17	4
18	3
19	4

Appendix F: Variable-Length-Code LZW Compression

The Variable-Length-Code LZW Compression is a variation of the Lempel-Ziv Compression algorithm in which variable-length codes are used to replace patterns detected in the original data. The algorithm uses a code or translation table constructed from the patterns encountered in the original data; each new pattern is entered into the table and its index is used to replace it in the compressed stream.

The compressor takes the data from the input stream and builds a code or translation table with the patterns as it encounters them; each new pattern is entered into the code table and its index is added to the output stream; when a pattern is encountered which had been detected since the last code table refresh, its index from the code table is put on the output stream, thus achieving the data compression. The expander takes input from the compressed data stream and builds the code or translation table from it; as the compressed data stream is processed, codes are used to index into the code table and the corresponding data is put on the decompressed output stream, thus achieving data decompression. The details of the algorithm are explained below. The Variable-Length-Code aspect of the algorithm is based on an initial code size (LZW-initial code size), which specifies the initial number of bits used for the compression codes. When the number of patterns detected by the compressor in the input stream exceeds the number of patterns encodable with the current number of bits, the number of bits per LZW code is increased by one.

The Raster Data stream that represents the actual output image can be represented as:

```
 7 6 5 4 3 2 1 0
 +---------------+
| LZW code size |
 +---------------+

 +---------------+ ----+
| block size | |
 +---------------+  |
| | +-- Repeated as many
| data bytes | | times as necessary.
```

```
|||
 +---------------+ ----+

 . . . . . . ------- The code that terminates the LZW
   compressed data must appear before
   Block Terminator.
 +---------------+
|0 0 0 0 0 0 0 0| Block Terminator
 +---------------+
```

The conversion of the image from a series of pixel values to a transmitted or stored character stream involves several steps. In brief these steps are:

1. Establish the Code Size - Define the number of bits needed to represent the actual data.
2. Compress the Data - Compress the series of image pixels to a series of compression codes.
3. Build a Series of Bytes - Take the set of compression codes and convert to a string of 8-bit bytes.
4. Package the Bytes - Package sets of bytes into blocks preceded by character counts and output.

Establish Code Size

The first byte of the Compressed Data stream is a value indicating the minimum number of bits required to represent the set of actual pixel values. Normally this will be the same as the number of color bits. Because of some algorithmic constraints however, black & white images which have one color bit must be indicated as having a code size of 2. This code size value also implies that the compression codes must start out one bit longer.

Compression

The LZW algorithm converts a series of data values into a series of codes which may be raw values or a code designating a series of values. Using text characters as an analogy, the output code consists of a character or a code representing a string of characters.

The LZW algorithm used in GIF matches algorithmically with the standard LZW algorithm with the following differences:

1. A special Clear code is defined which resets all compression/decompression parameters and tables to a start-up state. The value of this code is 2**<code size>. For example if the code size indicated was 4 (image was 4 bits/pixel) the Clear code value would be 16 (10000 binary). The Clear code can appear at any point in the image data stream and therefore requires the LZW algorithm to process succeeding codes as if a new data stream was starting. Encoders should output a Clear code as the first code of each image data stream.

2. An End of Information code is defined that explicitly indicates the end of the image data stream. LZW processing terminates when this code is encountered. It must be the last code output by the encoder for an image. The value of this code is <Clear code>+1.

3. The first available compression code value is <Clear code>+2.

4. The output codes are of variable length, starting at <code size>+1 bits per code, up to 12 bits per code. This defines a maximum code value of 4095 (0xFFF). Whenever the LZW code value would exceed the current code length, the code length is increased by one. The packing/unpacking of these codes must then be altered to reflect the new code length.

Build 8-bit Bytes

Because the LZW compression used for GIF creates a series of variable length codes, of between 3 and 12 bits each, these codes must be reformed into a series of 8-bit bytes that will be the characters actually stored or transmitted. This provides additional compression of the image. The codes are formed into a stream of bits as if they were packed right to left and then picked off 8 bits at a time to be output.

Assuming a character array of 8 bits per character and using 5 bit codes to be packed, an example layout would be similar to:

```
 +---------------+
0 | | bbbaaaaa
 +---------------+
1 | | dcccccbb
 +---------------+
2 | | eeeedddd
```

```
 +---------------+
3 || ggfffffe
 +---------------+
4 || hhhhhggg
 +---------------+
...
 +---------------+
N ||
 +---------------+
```

Note that the physical packing arrangement will change as the number of bits per compression code change but the concept remains the same.

Package the Bytes

Once the bytes have been created, they are grouped into blocks for output by preceding each block of 0 to 255 bytes with a character count byte. A block with a zero byte count terminates the Raster Data stream for a given image. These blocks are what are actually output for the GIF image. This block format has the side effect of allowing a decoding program the ability to read past the actual image data if necessary by reading block counts and then skipping over the data.

Further Reading

[1] Ziv, J. and Lempel, A. : "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, May 1977.
[2] Welch, T. : "A Technique for High-Performance Data Compression", Computer, June 1984.
[3] Nelson, M.R. : "LZW Data Compression", Dr. Dobb's Journal, October 1989.

Appendix G: On-line Capabilities Dialogue

NOTE : This section is currently (10 July 1990) under revision; the information provided here should be used as general guidelines. Code written based on this information should be designed in a flexible way to accommodate any changes resulting from the revisions.

The following sequences are defined for use in mediating control between a GIF sender and GIF receiver over an interactive communications line. These sequences do not apply to applications that involve downloading of static GIF files and are not considered part of a GIF file.

GIF Capabilities Enquiry

The GIF Capabilities Enquiry sequence is issued from a host and requests an interactive GIF decoder to return a response message that defines the graphics parameters for the decoder. This involves returning information about available screen sizes, number of bits/color supported and the amount of color detail supported. The escape sequence for the GIF Capabilities Enquiry is defined as:

```
ESC[>0g 0x1B 0x5B 0x3E 0x30 0x67
```

GIF Capabilities Response

The GIF Capabilities Response message is returned by an interactive GIF decoder and defines the decoder's display capabilities for all graphics modes that are supported by the software. Note that this can also include graphics printers as well as a monitor screen. The general format of this message is:

```
#version;protocol{;dev, width, height, color-bits, color-res}...<CR>
```

'#' GIF Capabilities Response identifier character
version GIF format version number; initially '87a'.

protocol='0' No end-to-end protocol supported by decoder Transfer as direct 8-bit data stream.

protocol='1' Can use CIS B+ error correction protocol to transfer GIF data interactively from the host directly to the display.

dev = '0' Screen parameter set follows.

dev = '1' Printer parameter set follows.

width Maximum supported display width in pixels.

height Maximum supported display height in pixels.

color-bits Number of bits per pixel supported. The number of supported colors is therefore 2**color-bits.

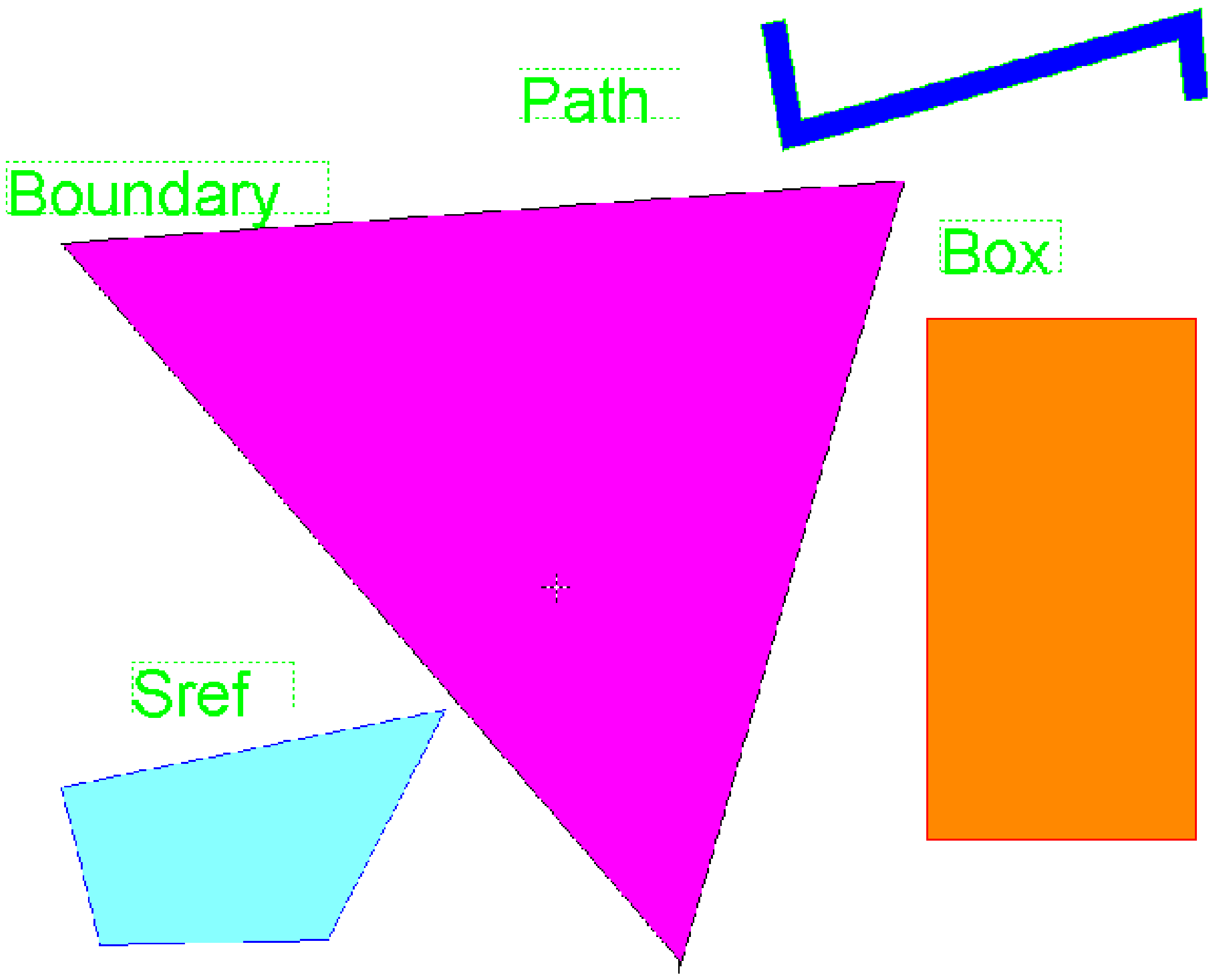
color-res Number of bits per color component supported in the hardware color palette. If color-res is '0' then no hardware palette table is available.

Note that all values in the GIF Capabilities Response are returned as ASCII decimal numbers and the message is terminated by a Carriage Return character.

GDSII format

INDEX

example



1. text presentation of GDSII file in KEYformat
2. hex presentation of same file
3. GDSII file

introduction

GDSII Stream format is the standard file format for transfering/archiving 2D graphical design data. It contains a hiearchy of structures, each structure containing elements (boundary/polygon, path/polyline, text,box, structure references, structure array references). The elements are situated on layers. It is a binary format that is platform independent, because it uses internally defined formats for its data types. While reading GDSII files, the GDSII internal data types (like reals, integers etc.) need to be converted to the platform/CAE package datatypes that are used.The GDSII format is a sequential list of records, each record contains a header to tell what information is in the record.The order of the record needs to be according to the GDSII BNF, because of this strict organization it is relativly easy to parse. The maximum number of vertixes is officially only 200 x,y pairs, but many packages can read up to the absolute maximum of 64k/2=32k, simple because this is the maximum record lenght that can be specified (two bytes).The format is hard to read, since it is binary, for that viewers are available to view (boolean) the contents as ASCII. Also an ASCII format has been developed (KEY format) which is more than just a text representation. It is possible to convert GDSIIformat to KEYformat and back. KEYformat has extended the basic primitives to contain cicrles, arcs, polygons/polylines with arc segments.

Bachus Nauer Forms

The Bachus Nauer Form uses the following symbols:

Symbol Name	Symbol	Meaning
Double Colon	::	"Is composed of."
Square brackets	[ ]	An element which can occor zero or one time.
Braces	{ }	Choose one of the elements within the braces.
Braces with an asteriks	{ }*	The elements within the braces can occur zero or more times.
Braces with a plus	{ }+	The elements within braces must occur one or more times.
Angle brackets	< >	These elements are further defined as a seperate entitie in the syntax list.
Vertical bar	|	Or

GDSII BNF

The following is the Bachus Naur Form of the GDSI format, the words in capital are the names of *RECORDS*

<stream format>	::=	HEADER BGNLIB LIBNAME [REFLIBS] [FONTS] [ATTRTABLE] [GENERATIONS] [<FormatType>] UNITS {<structure>}* ENDLIB
<FormatType>	::=	FORMAT | FORMAT {MASK}+ ENDMASKS
<structure>	::=	BGNSTR STRNAME [STRCLASS] {<element>}* ENDSTR
<element>	::=	{<boundary> | <path> | <sref> | <aref> | <text> | <node> | <box>} {<property>}* ENDEL
<boundary>	::=	BOUNDARY [ELFLAGS] [PLEX] LAYER DATATYPE XY
<path>	::=	PATH [ELFLAGS] [PLEX] LAYER DATATYPE [PATHTYPE][WIDTH] XY
<sref>	::=	SREF [ELFLAGS] [PLEX] SNAME [<strans>] XY
<aref>	::=	AREF [ELFLAGS] [PLEX] SNAME [<strans>] COLROW XY
<text>	::=	TEXT [ELFLAGS] [PLEX] LAYER <textbody>
<node>	::=	NODE [ELFLAGS]. [PLEX] LAYER NODETYPE XY
<box>	::=	BOX [ELFLAGS] [PLEX] LAYER BOXTYPE XY
<textbody>	::=	TEXTYPE [PRESENTATION] [PATHTYPE] [WIDTH] [<strans>] XY STRING
<strans>	::=	STRANS [MAG] [ANGLE]
<property>	::=	PROPATTR PROPVALUE

Record header

The Stream format output file is composed of variable length records. Record length is measured in bytes. The minimum record length is four bytes. Within the record, two bytes (16 bits) is a word. The 16 bits in a word are numbered 0 to 15, left to right.The first four bytes of a record compose the recordheader. The first two bytes of the recordheader contain a count (in eight-bit bytes) of the total record length, so the maximum length is 65536 (64k). The next record starts immediately after the last byte of the previous record.The third byte of the header is the record type. The fourth byte of the header identifies the type of data contained within the record. The fifth until count bytes of a record contain the data.

Bitnr	0	1	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Word1	Total Record length in bytes															
Word2	*Record Type*								*Data type*							
Word3	Data until Word n (total record length/2)															

Data Types

The fourth byte in the record header contains the data type for the rest of the record. The record length is used to find the number of items of the specified datatype.

Data Type	Value
No Data	0
Bit Array	1
Two Byte Signed Integer	2
Four Byte Signed Integer	3
Four Byte Real	4 (not used)
Eight Byte Real	5
ASCII string	6

1. **Bit Array:**
   A bit array is a word which uses the value of a particular bit or group of bits to represent data. A bit array allows oneword to represent a number of simple pieces of information.

2. **Two-Byte Signed Integer:**
   2-byte integer = 1 word 2s-complement representation. The range of two-byte signed integers is -32,768 to 32,767.

   The following is a representation of a two-byte integer, where S is the sign and M is the magnitude.

   smmmmmmm mmmmmmmm

   The following are examples of two-byte integers:

   00000000 00000001 = 1
   00000000 00000010 = 2
   00000000 10001001 = 137
   11111111 11111111 = -1
   11111111 11111110 = -2
   11111111 01110111 = -137

3. **Four-Byte Signed Integer:**
   4-byte integer = 2 word 2s-complement representation

   The range of four-byte signed integers is -2,147,483,648 to 2,147,483,647.

   The following is a representation of a four-byte integer, where S is the sign and M is the magnitude.

   smmmmmmm mmmmmmmm mmmmmmmm mmmmmmmm

   The following are examples of four-byte integers:

   00000000 00000000 00000000 00000001 = 1

00000000 00000000 00000000 00000010 = 2
00000000 00000000 00000000 10001001 = 137
11111111 11111111 11111111 11111111 = -1
11111111 11111111 11111111 11111110 = -2
11111111 11111111 11111111 01110111 = -137

4. **Four-Byte Real**
   4-byte real = 2-word floating point representation

   (See 5.)

5. **Eight-Byte Real**
   8-byte real = 4-word floating point representation

   For all non-zero values:
   - A floating point number has three parts: the sign, the exponent, and the mantissa.
   - The value of a floating point number is defined as:
   - (Mantissa) x (16 raised to the true value of the exponent field).
   - The exponent field (bits 1-7) is in Excess-64 representation.
   - The 7-bit field shows a number that is 64 greater than the actual exponent.
   - The mantissa is always a positive fraction >=1/16 and <1. For a 4-byte real, the mantissa is bits 8-31. For an 8-byte real, the mantissa is bits 8-63.
   - The binary point is just to the left of bit 8.
   - Bit 8 represents the value 1/2, bit 9 represents 1/4, etc.
   - In order to keep the mantissa in the range of 1/16 to 1, the results of floating point arithmetic are normalized. Normalization is a process where by the mantissa is shifted left one hex digit at a time until its left FOUR bits represent a non-zero quantity. For every hex digit shifted, the exponent is decreased by one. Since the mantissa is shifted four bits at a time, it is possible for the left three bits of the normalized mantissa to be zero. A zero value, also called true zero, is represented by a number with all bits zero.

The following are representations of 4-byte and 8-byte reals, where S is the sign, E is the exponent, and M is the magnitude. Examples of 4-byte reals are included in the following pages, but 4-byte reals are not used currently. The representation of the negative values of real numbers is exactly the same as the positive, except that the highest order bit is 1, not 0. In the eight-byte real representation, the first four bytes are exactly the same as in the four-byte real representation. The last four bytes contain additional binary places for more resolution.

4-byte real:

SEEEEEEE MMMMMMMM MMMMMMMM MMMMMMMM

8-byte real:

SEEEEEEE MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM

Examples of 4-byte real:

Note: In the first six lines of the following example, the 7-bit exponent field = 65. The actual exponent is 65-64=1.

01000001 00010000 00000000 00000000 = 1
01000001 00100000 00000000 00000000 = 2
01000001 00110000 00000000 00000000 = 3
11000001 00010000 00000000 00000000 = -1
11000001 00100000 00000000 00000000 = -2
11000001 00110000 00000000 00000000 = -3
01000000 10000000 00000000 0000000 = 0 .5
01000000 10011001 10011001 1001100 = 1 .6
01000000 10110011 00110011 0011001 = 1 .7
01000001 00011000 00000000 00000000 = 1.5
01000001 00011001 10011001 10011001 = 1.6
01000001 00011011 00110011 00110011 = 1.7
00000000 00000000 00000000 00000000 = 0
01000001 00010000 00000000 00000000 = 1
01000001 10100000 00000000 00000000 = 10
01000010 01100100 00000000 00000000 = 100
01000011 00111110 00000001 00000000 = 1000
01000100 00100111 00010000 00000000 = 10000
01000101 00011000 01101010 00000000 = 100000

6. **ASCII String**
A collection of ASCII characters, where each character is represented by one byte. All odd length strings must be padded with a null character (the number zero), and the byte count for the record containing the ASCII string must include this null character. Stream read-in programs must look for the null character and decrease the length of the string by one if the null character is present.

Record Types Overview

The following table gives an overview of all the record that are used within a GDSII file.

Nr.	Code	Mnemonic	Data Type	description
0	0002	HEADER	Two-Byte Signed Integer	version number
1	0102	BGNLIB	Two-Byte Signed Integer	begin of library, last modification date and time
2	0206	LIBNAME	Two-Byte Signed Integer	name of library
3	0305	UNITS	Eight-Byte Real	user and database units
4	0400	ENDLIB	No Data	end of library
5	0502	BGNSTR	Two-Byte Signed Integer	begin of structure + creation and modification time
6	0606	STRNAME	ASCII string	name of structure

7	0700	ENDSTR	No Data	end of structure
8	0800	BOUNDARY	No Data	begin of boundary element
9	0900	PATH	No Data	begin of path element
10	0A00	SREF	No Data	begin of structure reference element
11	0B00	AREF	No Data	begin of array reference element
12	0C00	TEXT	No Data	begin of text element
13	0D02	LAYER	Two-Byte Signed Integer	layer number of element
14	0E02	DATATYPE	Two-Byte Signed Integer	Datatype number of element
15	0F03	WIDTH	Four-Byte Signed Integer	width of element in db units
16	1003	XY	Four-Byte Signed Integer	list of xy coordinates in db units
17	1100	ENDEL	No Data	end of element
18	1206	SNAME	ASCII string	name of structure reference
19	1302	COLROW	Two-Byte Signed Integer	number of colomns and rows in array reference
21	1500	NODE	No Data	begin of node element
22	1602	TEXTTYPE	Two-Byte Signed Integer	texttype number
23	1701	PRESENTATION	Bit Array	text presentation, font
25	1906	STRING	ASCII string	character string for text element
26	1A01	STRANS	Bit Array	array reference, structure reference and text transform flags
27	1B05	MAG	Eight Byte Real	magnification factor for text and references
28	1C05	ANGLE	Eight Byte Real	rotation angle for text and references
31	1F06	REFLIBS	ASCII string	name of referenced libraries
32	2006	FONTS	ASCII string	name of text fonts definition files
33	2102	PATHTYPE	Two-Byte Signed Integer	type of PATH element end (rounded, square)
34	2202	GENERATIONS	Two-Byte Signed Integer	number of deleted structure ?????
35	2306	ATTRTABLE	ASCII string	attribute table, used in combination with element properties
38	2601	ELFLAGS	Two-Byte Signed Integer	template data
42	2A02	NODETYPE	Two-Byte Signed Integer	node type number for NODE element

43	2B02	PROPATTR	Two-Byte Signed Integer	attribute number
44	2C06	PROPVALUE	ASCII string	attribute name
45	2D00	BOX	No Data	begin of box element
46	2E02	BOXTYPE	Two-Byte Signed Integer	boxtype for box element
47	2F03	PLEX	Four-Byte Signed Integer	plex number
50	3202	TAPENUM	Two-Byte Signed Integer	Tape Number
51	3302	TAPECODE	Two-Byte Signed Integer	Tape code
54	3602	FORMAT	Two-Byte Signed Integer	format type
55	3706	MASK	ASCII string	list of layers
56	3800	ENDMASKS	No Data	end of MASK

Record types description

Records are always an even number of bytes long. The first four bytes of a record are the *record header*. If a record contains ASCII string data and the ASCII string is an odd number of bytes long, the data is padded with a null character. This paragraph lists the record types with a brief description of each. The descriptions include the record name and a four-digit number in brackets. The first two numbers within the brackets are the record type, and the last two numbers in brackets are the data type. All record numbers are expressed in hexadecimal.

0	HEADER	0002	Two-Byte Signed Integer

Contains two bytes of data representing the Stream version number.

1	BGNLIB	0102	Two-Byte Signed Integer

Contains the last modification time of a library (two bytes each for year, month, day, hour, minute, and second), the time of last access (same format), and marks the beginning of a library.

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
word1	l C (hex) # of bytes in record															
word2	01 (hex) 02 (hex)															
word3	year (lastmodification time)															
word4	month															
word5	day															
word6	hour															
word7	minute															
word8	second															
word9	year (last access time)															
word10	month															
word11	day															
word12	hour															
word13	minute															
word14	second															

2	LIBNAME	0206	ASCII String

Contains a string which is the library name. The library name must follow UNIX filename conventions for length and valid characters. The library name may include the file extension (.sf or db in most cases).

3	UNITS	0305	Eight-Byte Real

Contains two eight-byte real numbers. The first number is the size of a database unit in user units. The second number is the size of a database unit in meters. For example, if you create a library with the default units (user unit = 1 micron and 1000 database units per user unit), the first number is .001, and the second number is 1E-9. Typically, the first number is less than 1, since you use more than 1 database unit per user unit. To calculate the size of a user unit in meters, divide the second number by the first.

4	ENDLIB	0400	No Data

Marks the end of a library.

5	BGNSTR	0502	Two-Byte Signed Integer

Contains the creation time and last modification time of a structure (in the same format as the BGNLIB record), and marks the beginning of a structure.

6	STRNAME	0606	ASCII String

Contains a string which is the structure name. A structurename may be up to 32 characters long. Legal structurename characters are:

- A through Z
- a through z
- 0 through 9
- Underscore (_)
- Question mark (?)
- Dollar sign ($)

7	ENDSTR	0700	No Data

Marks the end of a structure.

8	BOUNDARY	0800	No Data

Marks the beginning of a boundary element.

9	PATH	0900	No Data

Marks the beginning of a path element.

10	SREF	0A00	No Data

Marks the beginning of an Sref (structure reference) element.

11	AREF	0B00	No Data

Marks the beginning of an Aref (array reference) element.

12	TEXT	0C00	No Data

Marks the beginning of a text element.

13	LAYER	0D02	Two-Byte Signed Integer

Contains two bytes which specify the layer. The value of the layer must be in the range of 0 to 255.

14	DATATYPE	OEO2	Two-Byte Signed Integer

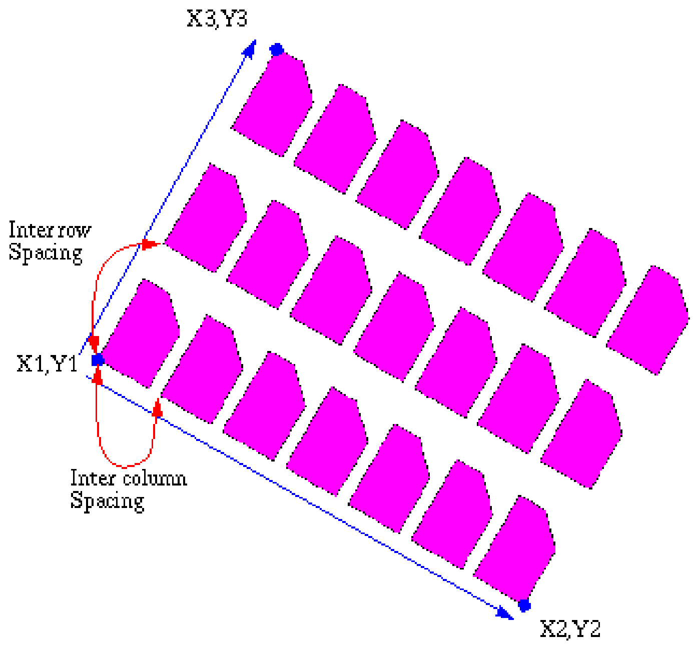
Contains two bytes which specify the datatype. The value of the datatype must be in the range of 0 to 255.

15	WIDTH	0F03	Two-Byte Signed Integer

Contains four bytes which specify the width of a path or text lines in database units. A negative value for width means that the width is absolute, that is, the width is not affected by the magnification factor of any parent reference. If omitted, zero is assumed.

16	XY	1003	Two-Byte Signed Integer

- Contains an array of XY coordinates in database units. Each X or Y coordinate is four bytes long. Path elements may have a minimum of 2 and a maximum of 200 coordinates. Boundary and border elements may have a minimum of 4 and a maximum of 200 coordinates. The first and last coordinates of a boundary or border must coincide.
- A text, or Sref element may have only one coordinate.
- An Aref has exactly three coordinates. In an Aref, the first coordinate is the array reference point (origin point). The other two coordinates are already rotated, reflected as specified in the STRANS record (if specified). So in order to calculate the intercolomn and interrow spacing, the coordinates must be mapped back to their original position, or the vector lenght (x1,y1-> x3,y3) must be divided by the number of row etc. . The second coordinate locates a position which is displaced from the reference point by the inter-column spacing times the number of columns. The third coordinate locates a position which is displaced from the reference point by the inter-row spacing times the number of rows. For an example of an array lattice see the next picture.



Aref rotated -30 degrees.

- A node may have from one to 50 coordinates.
- A box must have five coordinates, with the first and last coordinates being the same.

17	ENDEL	1100	No Data

Marks the end of an element.

18	SNAME	1206	ASCII string

Contains the name of a referenced structure.See also STRNAME.

19	COLROW	1302	Two-Byte Signed Integer

Contains four bytes. The first two bytes contain the number of columns in the array. The third and fourth bytes contain the number of rows. Neither the number of columns nor the number of rows may exceed

32,767 (decimal), and both are positive. See also AREF.

21	NODE	1500	No Data

Present Marks the beginning of a node

22	TEXTTYPE	1602	Two-Byte Signed Integer

Contains two bytes representing texttype. The value of the texttype must be in the range 0 to 255.

23	PRESENTATION	1701	Bit Array

Contains one word (two bytes) of bit flags for text presentation. Bits 10 and 11, taken together as a binary number, specify the font (00 means font 0, 01 rneans font 1, 10 means font 2, and 11 means font 3). Bits 12 and 13 specify the vertical justification (00 means top, 01 means middle, and 10 means bottom). Bits 14 and 15 specify the horizontal justification (00 means left, 01 means center, and 10 means right). Bits 0 through 9 are reserved for future use and must be cleared. If this record is omitted, then top-left justification and font 0 are assumed. The following shows a PRESENTATION record.

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
word1	6 (hex) # of bytes in record															
word2	17 (hex)								01 (hex)							
word3	unused										font number		vertical presentaion		horizontal presentation	

25	STRING	1906	ASCII String

Contains a character string, up to 512 characters long, for text presentation.

26	STRANS	1A01	Bit Array

Contains two bytes of bit flags for Sref, Aref, and text transfornnation. Bit 0 (the leftmost bit) specifies reflection. If bit 0 is set, the element is reflected about the X-axis before angular rotation. For an Aref, the entire array is reflected, with the individual array members rigidly attached. Bit 13 flags absolute magnification. Bit 14 flags absolute angle. Bit 15 (the rightmost bit) and all remaining bits are reserved for future use and must be cleared. If this record is omitted, the element is assumed to have no reflection, non-absolute magnification, and non- absolute angle.
The following shows a STRANS record.

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
word1	6 (hex) # of bytes in record															
word2	1A (hex)								01 (hex)							
word3	reflection	unused												absolute magnification	absolute angle	unused

27	MAG	1B05	Eight Byte Real

Eight-Byte Real Contains a double-precision real number (8 bytes), which is the magnification factor. If this record is omitted, a magnification factor of one is assumed.

28	ANGLE	1C05	Eight Byte Real

Eight-Byte Real Contains a double-precision real number (8 bytes), which is the angular rotation factor. The angle of rotation is measured in degrees and in the counterclockwise direction. For an Aref, the ANGLE rotates the entire array (with the individual array members rigidly attached) about the array reference point. For COLROW record information, the angle of rotation is already inlcuded in the coordinates. If this record is omitted, an angle of zero degrees is assumed.

31	REFLIBS	1F06	ASCII String

Contains the names of the reference libraries. This record must be present if any reference libraries are bound to the working library. The name of the first reference library starts at byte 5 (immediately following the record header) and continues for 44 bytes. The next 44 bytes contain the name of the second library. The record is extended by 44 bytes for each additional library (up to 15) which is bound for reference. The reference library names may include directory specifiers (separated with "/") and an extension (separated with "."). If either the first or second library is not named, its place is filled with nulls.

32	FONTS	2006	ASCII String

Contains the names of the textfont definition files. This record must be present if any of the four fonts have acorresponding textfont definition file. This record must not be present if none of the fonts have a textfont definition file. The textfont filename of font 0 starts the record, followed by the textfont files of the remaining three fonts.Each filename is 44 bytes long. The filename is padded withnulls if the name is shorter than 44 bytes. The filename is null if no textfont definition corresponds to the font. The textfont filenames may include directory specifiers (separated with "/" and an extension (separated with ".").

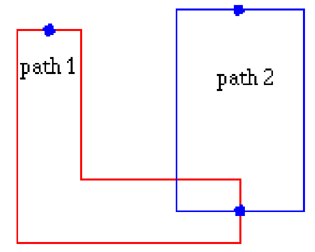
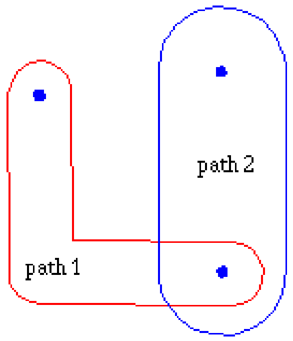
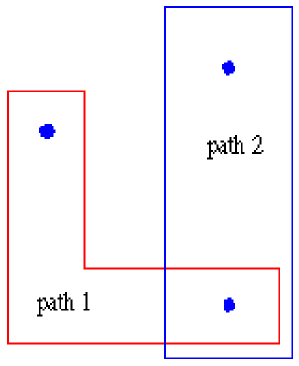
33	PATHTYPE	2102	Two-Byte Signed Integer

This record contains a value that describes the type of path endpoints. The value is

- 0 for square-ended paths that endflush with their endpoints
- 1 for round-ended paths
- 2 for square-ended paths that extend a half-width beyond their endpoints

If not specified, a Path-type of 0 is assumed.

The following picture shows the pathtypes

	Pathtype 0 produces a square-ended path, ending flush with thedigitized endpoints. This is the de-fault pathtype if none is specified
	Pathtype 1 produces a round-ended path. The two ends aresemicircular with center at thedigitized endpoints.
	Pathtype 2 produces a square-ended path. The ends of the pathextend beyond the digitized end-points by one-half the path width.

34	GENERATIONS	2202	Two-Byte Signed Integer

This record contains a value to indicate the number of copies of deleted or back-up structures to retain. This numbermust be at least 2 and not more than 99. If the GENERATION record is omitted, a value of 3 is assumed.

35	ATTRTABLE	2306	Two-Byte Signed Integer

Contains the name of the attribute definition file. This record is present only if an attribute definition file is bound to the library. The attribute defenition filename may include directory specifiers (separated with "/") and an extension (separated with "."). Maximum record size is 44 bytes.

36	STYPTABLE	2502	Two-Byte Signed Integer

Unrelesed Feature

37	STRTYPE	2502	Two-Byte Signed Integer

Unrelesed Feature

38	ELFLAGS	2601	Bit Array

Contains two bytes of bit flags. Bit 15 (the rightmost bit)specifies Template data. Bit 14 specifies External data(also referred to as Exterior data). All other bits are currently unused and must be cleared to 0. If this record isomitted, all bits are assumed to be 0. The following shows an ELFLAGS record.

For additional information on Template data, consult the GDSII Reference Manual. For additional information on External data, consult the CustomPlus User’s Manual.

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
word1	6 (hex) # of bytes in record															
word2	26 (hex)								01 (hex)							
word3	unused														external data	template data

39	ELKEY	2703	Two-Byte Signed Integer

(Unreleased feature)

40	LINKTYPE	28	Two-Byte Signed Integer

(Unreleased feature)

41	LINKKEYS	29	Two-Byte Signed Integer

(Unreleased feature)

42	NODETYPE	2A02	Two-Byte Signed Integer

Contains two bytes which specify nodetype. The value ofthe nodetype must be in the range of 0 to 255.

43	PROPATTR	2B02	Two-Byte Signed Integer

Contains two bytes which specify the attribute number. The attribute number is an integer from 1 to 127. Attribute numbers 126 and 127 are reserved for the user integer and userstring (CSD) properties which existed prior to Release 3.0.

44	PROPVALUE	2C06	ASCII string

Contains the string value associated with the attribute named in the preceding PROPATTR record. Maximumlength is 126 characters. The attribute-value pairs associated with any one element must all have distinct attribute numbers. Also, the total amount of property data that may be associated with any one element is limited: thetotal length of all the strings, plus twice the number of attribute-value pairs, must not exceed 128 (or 512 if the element is an Sref, Aref, contact, nodeport, or node).

For example, if a boundary element uses property attribute2 with property value "metal," and property attribute 10 with property value "property," the total amount of property data is 18 bytes. This is 6 bytes for "metal" (odd-length strings must be padded with a null) + 8 for "property" + 2 times the 2 attributes (4) = 18.

45	BOX	2D00	No Data

Marks the beginning of a box element.

46	BOXTYPE	2E02	Two-Byte Signed Integer

Contains two bytes which specify boxtype. The value of the boxtype must be in the range of 0 to 255.

47	PLEX	2F03	Two-Byte Signed Integer

A unique positive number which is common to all elementsof the plex to which this element belongs. The head of the plex is flagged by setting the seventh bit; therefore, plexnumbers should be small enough to occupy only the right-most 24 bits. If this record is not present, the element is not a plex member Applies to Pathtype 4.Contains four bytes which specify indatabase units the distance a path outline begins before orafter the last point of the path. Value can be negative.

50	TAPENUM	3202	Two-Byte Signed Integer

Contains two bytes which specify the number of the current reel of tape for a multi-reel Stream file. For the first tape, the TAPENUM is 1: for the second tape, the TAPENUM is 2. For each additional tape, increment the TAPENum by one.

51	TAPECODE	3302	Two-Byte Signed Integer

Contains 12 bytes. This is a unique 6-integer code which iscommon to all the reels of multi-reel Stream file. It verifies that the correct reels are being read.

52	STRCLASS	3401	Two-Byte Signed Integer

Not used

53	RESERVED	3503	Two-Byte Signed Integer

This record type was used for NUMTYPES but was not required.

54	FORMAT	3602	Two-Byte Signed Integer

Defines the format of a Stream tape in two bytes. The possible values are:

1. for GDSII Archive format
2. for GDSII Filtered format
3. for EDSM Archive format
4. for EDSHI Filtered fornat

An Archive Stream file contains elements for all the layers and data types. In an Archive Stream file, the FORMAT record is followed immediately by the UNITS record. A file which does not have the

FORMAT record is assumed to be an Archive file.

A Filtered Stream file contains only the elements on the layers and with the datatypes you specify during creation ofthe Stream file. The list of layers and datatypes specified appear in MASK records. At least one MASK record must immediately follow the FORMAT record. The MASK records are terminated with the ENDMASKS record.

55	MASK	3706	ASCII string

(Required for and present only in FilteredStream file.)
Contains the list of layers and datatypes specified by the user when creating the file. At least one MASK record must immediately follow the FORMAT record. More than one MASK record may occur. The last MASK record is followed by the ENDMASK record.
In the MASK list, datatypes are separated from the layers with a semicolon. Individual layers or datatypes are sepa-rated with a space. A range of layers or datatypes is specified with a dash.

An example MASK list looks like this: 1 5 -7 10 ; 0- 255

56	ENDMASKS	3800	NoData

(Required for and present only in FilteredStream file.)
Marks the end of the MASK records. The ENDMASKS record must follow the last MASK record. ENDMASKS is immediately followed by the UNITS record.

Incomplete; check here!

MPEG Audio Layer I/II/III frame header

An MPEG audio file is built up from smaller parts called frames. Generally, frames are independent items. Each frame has its own header and audio informations. As there is no file header, you can cut any part of MPEG file and play it correctly (this should be done on frame boundaries but most applications will handle incorrect headers). However, for Layer III, this is not 100% correct. Due to internal data organization in MPEG Layer III files, frames are often dependent of each other and they cannot be cut off just like that.

When you want to read info about an MPEG file, it is usually enough to find the first frame, read its header and assume that the other frames are the same. But this may not be always the case. Variable bitrate MPEG files may use so called bitrate switching, which means that bitrate changes according to the content of each frame. This way lower bitrates may be used in frames where it will not reduce sound quality. This allows making better compression while keeping high quality of sound.

The frame header is constituted by the very first four bytes (32bits) in a frame. The first eleven bits (or first twelve bits, see below about frame sync) of a frame header are always set and they are called "frame sync". Therefore, you can search through the file for the first occurence of frame sync (meaning that you have to find a byte with a value of 255, and followed by a byte with its three (or four) most significant bits set). Then you read the whole header and check if the values are correct. You will see in the following table the exact meaning of each bit in the header. Each value that is specified as reserved, invalid, bad, or not allowed should indicate an invalid header.

Frames may have a CRC check. The CRC is 16 bits long and, if it exists, it follows the frame header. After the CRC comes the audio data. You may calculate the CRC of the frame, and compare it with the one you read from the file. This is actually a very good method to check the MPEG frame validity.

Here is a presentation of the header content. Characters from A to M are used to indicate different fields. In the table below, you can see details about the content of each field.

AAAAAAAA AAABBCCD EEEEFFGH IIJJKLMM

Sign	**Length (bits)**	**Position (bits)**	**Description**
A	11	(31-21)	Frame sync (all bits set)
B	2	(20,19)	MPEG Audio version ID

00 - MPEG Version 2.5 (unofficial)
01 - reserved
10 - MPEG Version 2 (ISO/IEC 13818-3)
11 - MPEG Version 1 (ISO/IEC 11172-3)

Note: MPEG Version 2.5 is not official standard. It is an extension of the standard used for very low bitrate files. If your decoder does not support this extension, it is recommended for you to use 12 bits for synchronization instead of 11 bits.

C 2 (18,17) Layer description
00 - reserved
01 - Layer III
10 - Layer II
11 - Layer I

D 1 (16) Protection bit
0 - Protected by CRC (16bit crc follows header)
1 - Not protected

E 4 (15,12) Bitrate index

bits	V1,L1	V1,L2	V1,L3	V2,L1	V2, L2 & L3
0000	free	free	free	free	free
0001	32	32	32	32	8
0010	64	48	40	48	16
0011	96	56	48	56	24
0100	128	64	56	64	32
0101	160	80	64	80	40
0110	192	96	80	96	48
0111	224	112	96	112	56
1000	256	128	112	128	64
1001	288	160	128	144	80
1010	320	192	160	160	96
1011	352	224	192	176	112
1100	384	256	224	192	128
1101	416	320	256	224	144
1110	448	384	320	256	160
1111	bad	bad	bad	bad	bad

NOTES: All values are in kbps
V1 - MPEG Version 1
V2 - MPEG Version 2 and Version 2.5
L1 - Layer I
L2 - Layer II
L3 - Layer III

"free" means free format. If the correct fixed bitrate (such

files cannot use variable bitrate) is different than those presented in upper table it must be determined by the application. This may be implemented only for internal purposes since third party applications have no means to findout correct bitrate. Howewer, this is not impossible to do but demands lots of efforts.
"bad" means that this is not an allowed value

MPEG files may have variable bitrate (VBR). Each frame may be created with different bitrate. It may be used in all layers. Layer III decoders must support this method. Layer I & II decoders may support it.

For Layer II there are some combinations of bitrate and mode which are not allowed. Here is a list of allowed combinations.

bitrate	single channel	stereo	intensity stereo	dual channel
free	yes	yes	yes	yes
32	yes	no	no	no
48	yes	no	no	no
56	yes	no	no	no
64	yes	yes	yes	yes
80	yes	no	no	no
96	yes	yes	yes	yes
112	yes	yes	yes	yes
128	yes	yes	yes	yes
160	yes	yes	yes	yes
192	yes	yes	yes	yes
224	no	yes	yes	yes
256	no	yes	yes	yes
320	no	yes	yes	yes
384	no	yes	yes	yes

F 2 (11,10) Sampling rate frequency index (values are in Hz)

bits	MPEG1	MPEG2	MPEG2.5
00	44100	22050	11025
01	48000	24000	12000
10	32000	16000	8000
11	reserv.	reserv.	reserv.

G 1 (9) Padding bit
0 - frame is not padded
1 - frame is padded with one extra slot
Padding is used to fit the bit rates exactly. For an example: 128k 44.1kHz layer II uses a lot of 418 bytes and some of 417 bytes long frames to get the exact 128k bitrate. For Layer I slot is 32 bits long, for Layer II and Layer III slot is 8 bits long.

H	1	(8)	Private bit. It may be freely used for specific needs of an application.
I	2	(7,6)	Channel Mode 00 - Stereo 01 - Joint stereo (Stereo) 10 - Dual channel (2 mono channels) 11 - Single channel (Mono) Note: Dual channel files are made of two independant mono channel. Each one uses exactly half the bitrate of the file. Most decoders output them as stereo, but it might not always be the case. One example of use would be some speech in two different languages carried in the same bitstream, and and appropriate decoder would decode only the choosen language </>

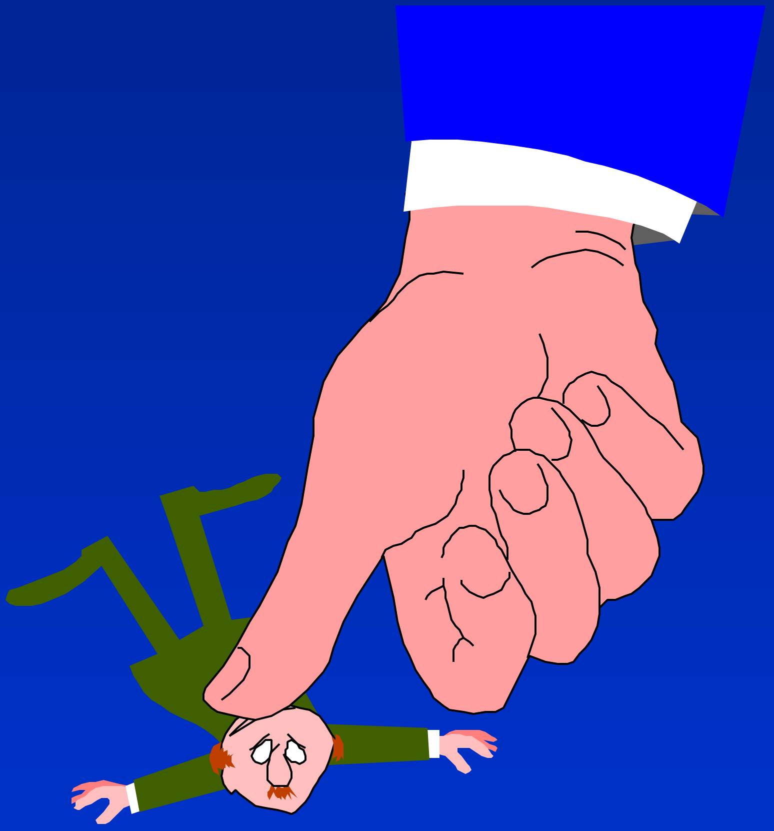
MPEG-2 Tutorial

Contents

In this discussion we will cover:

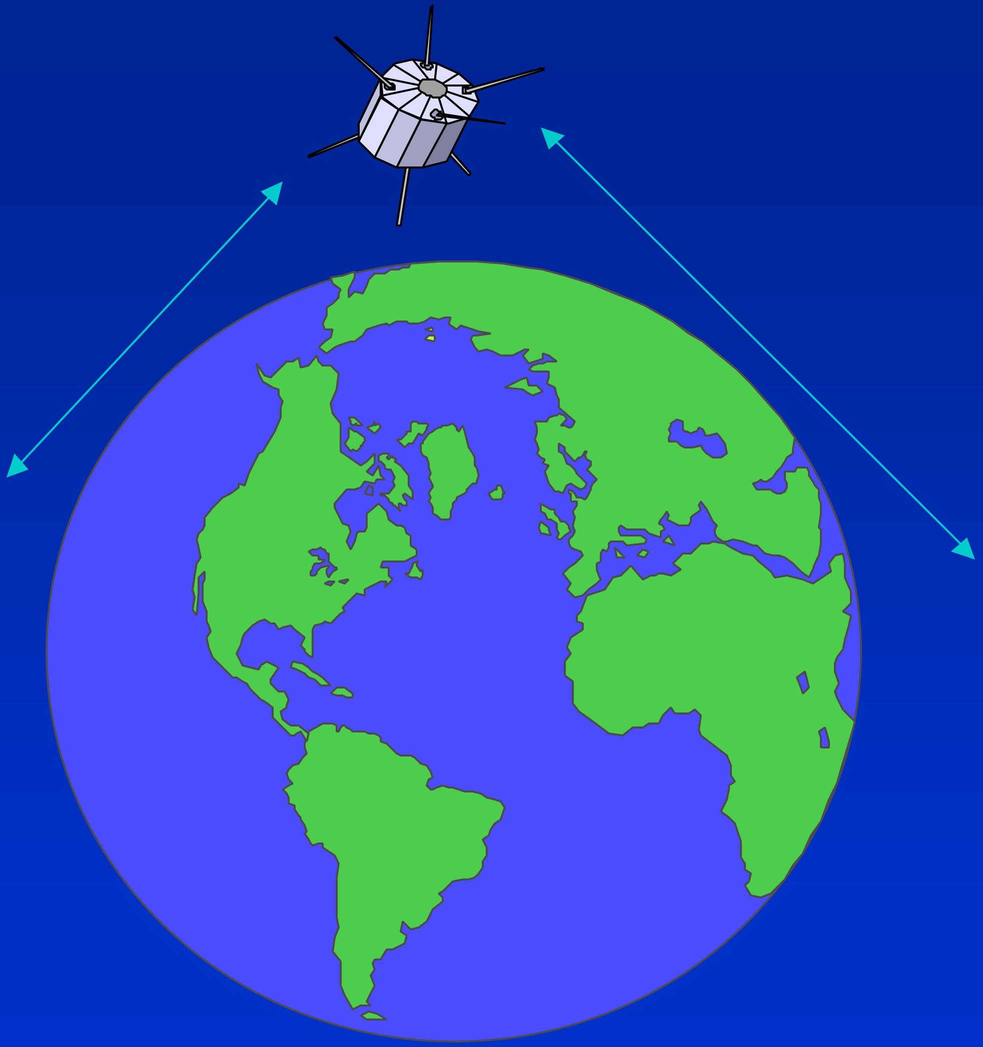
- **MPEG: Why?**
- **Overview of Compression**
- **MPEG-2 CODEC (Encoder/Decoder)**
- **Transport Layer**

Why Compress Data?



Less Data

- **Optimizing:**
  - **Cost**
  - **Bandwidth**
  - **Storage**
- **Enabling:**
  - **Higher Quality**
  - **More Channels**



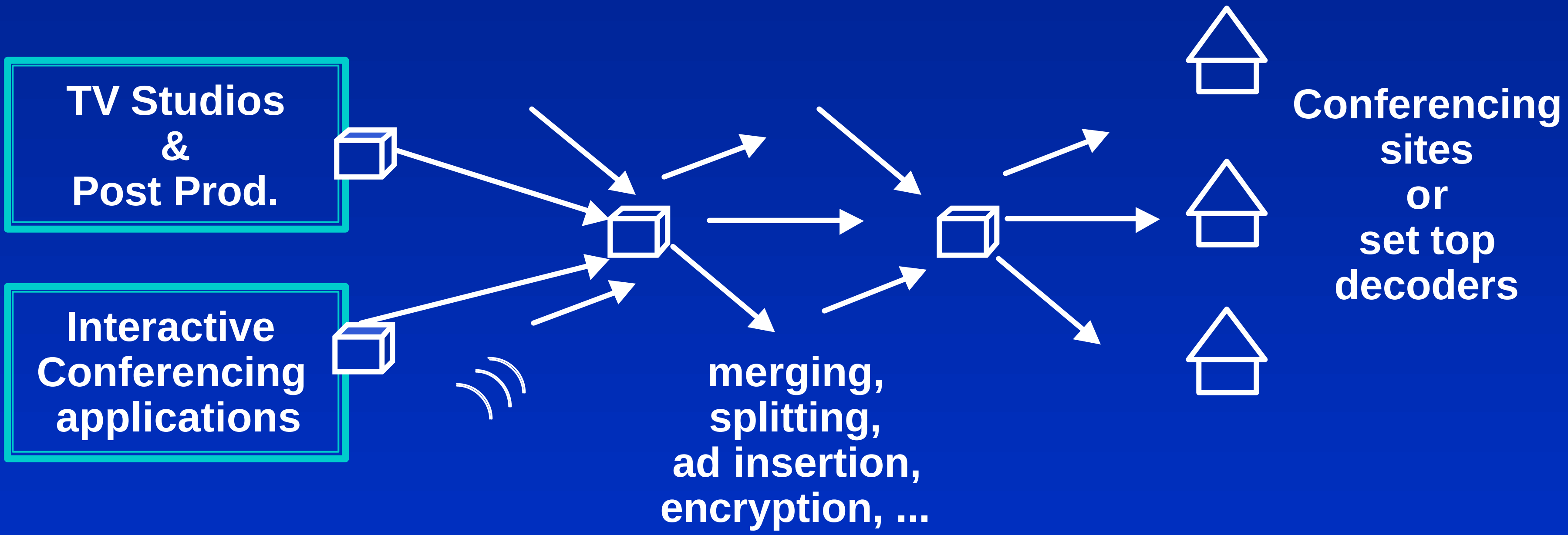
MPEG-2 Applications

BSS	Broadcasting Satellite Service (to the home)
CATV	Cable TV Distribution on optical networks, copper, etc.
CDAD	Cable Digital Audio Distribution
DAB	Digital Audio Broadcasting (terrestrial and satellite broadcasting)
DTTB	Digital Terrestrial Television Broadcast
EC	Electronic Cinema
ENG	Electronic News Gathering (including SNG, Satellite News Gathering)
FSS	Fixed Satellite Service (e.g. to head ends)
HTT	Home Television Theatre
IPC	Interpersonal Communications (videoconferencing, videophone, etc.)
ISM	Interactive Storage Media (optical disks, etc.)
MMM	Multimedia Mailing
NCA	News and Current Affairs
NDB	Networked Database Services (via ATM, etc.)
RVS	Remote Video Surveillance
SSM	Serial Storage Media (digital VTR, etc.)

Typical Video Data Rates

10-bit CCIR 601	270 Mbps
8-bit CCIR 601	216 Mbps
8-bit 601 (active only)	167 Mbps
Digital Betacam (R)	~90 Mbps
MPEG-2 4:2:2P@ML	10-50 Mbps
MPEG-2 MP@ML	2-15 Mbps
MPEG-1 constrain. param.	0.5-1.8 Mbps
H.261 videoconferencing	64 Kbps - 1.5 Mbps
H.263 videoconferencing	4 Kbps - 0.5 Mbps

MPEG Systems


TV Studios & Post Prod.
Interactive Conferencing applications
merging, splitting, ad insertion, encryption, ...
Conferencing sites or set top decoders

Digital Video Compression

- **Goal: Minimize video storage capacity or bandwidth. Measured in bits/second of video.**
- **What Determines the Bit Rate?**
  - **Picture format**
  - **Scene complexity**
  - **Constraints**
    - **Quality**
    - **Delay**
    - **Encoder complexity and algorithm**
  - **Noise**

MPEG Picture Coding Tools

- Reduced Resolution
  - Spatial: send small pictures
  - Temporal: reduce number of picture per second
- Transform Coding
  - Discrete Cosine Transform
- Subjective Importance
  - 4:2:2 and 4:2:0 chrominance formats
  - Quantization matrices

MPEG Picture Coding Tools

- Entropy Coding
  - Run length coding
  - Huffman coding
- Predictive Coding
  - Motion Estimation
  - DPCM (Discrete Pulse Code Modulation)

MPEG-2

- Start with CCIR-601 Video
- Serial Component Digital Video
- 270 Mbit Rate

IntRA-Frame Coding

I

Reduce Size

Picture Sizes

CCIR 601 525/30/2:1	720 x 486
CCIR 601 625/25/2:1	720 x 576
MPEG-2 422P@ML 30fps	720 x 512
MPEG-2 422P@ML 25fps	720 x 608
MPEG-2 30fps (quasi-std)	704 x 480
MPEG-2 25fps (quasi-std)	704 x 576
SIF (30fps, 25fps)	352 x 240,288
CIF (always 30fps)	352 x 240
HHR, 2/3-HR, 3/4-HR	352,480,528 x 480,576
QSIF (30fps, 25fps)	176 x 128,144
QCIF (always 30fps)	176 x 144

IntRA-Frame Coding

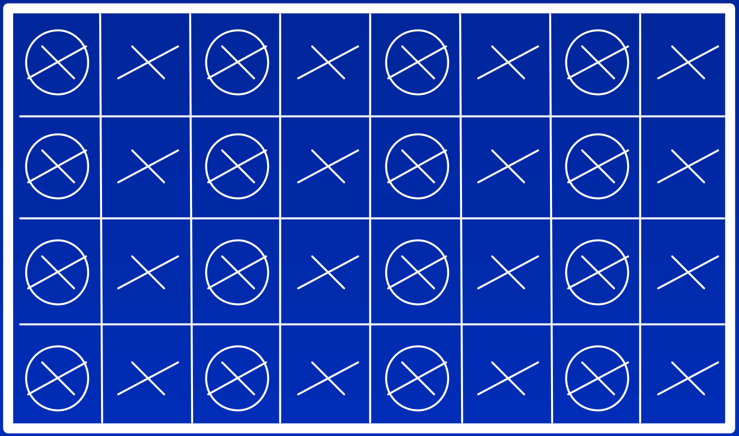
I

Reduce Size

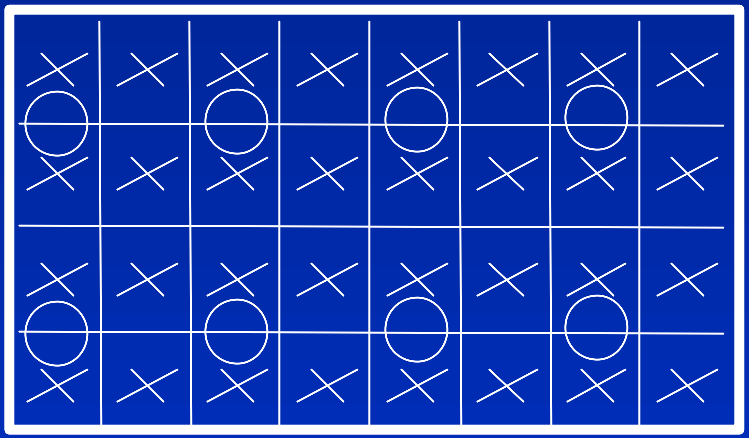
II

Sub-sample

4:2:0 Chroma Sub-Sampling



4:2:2

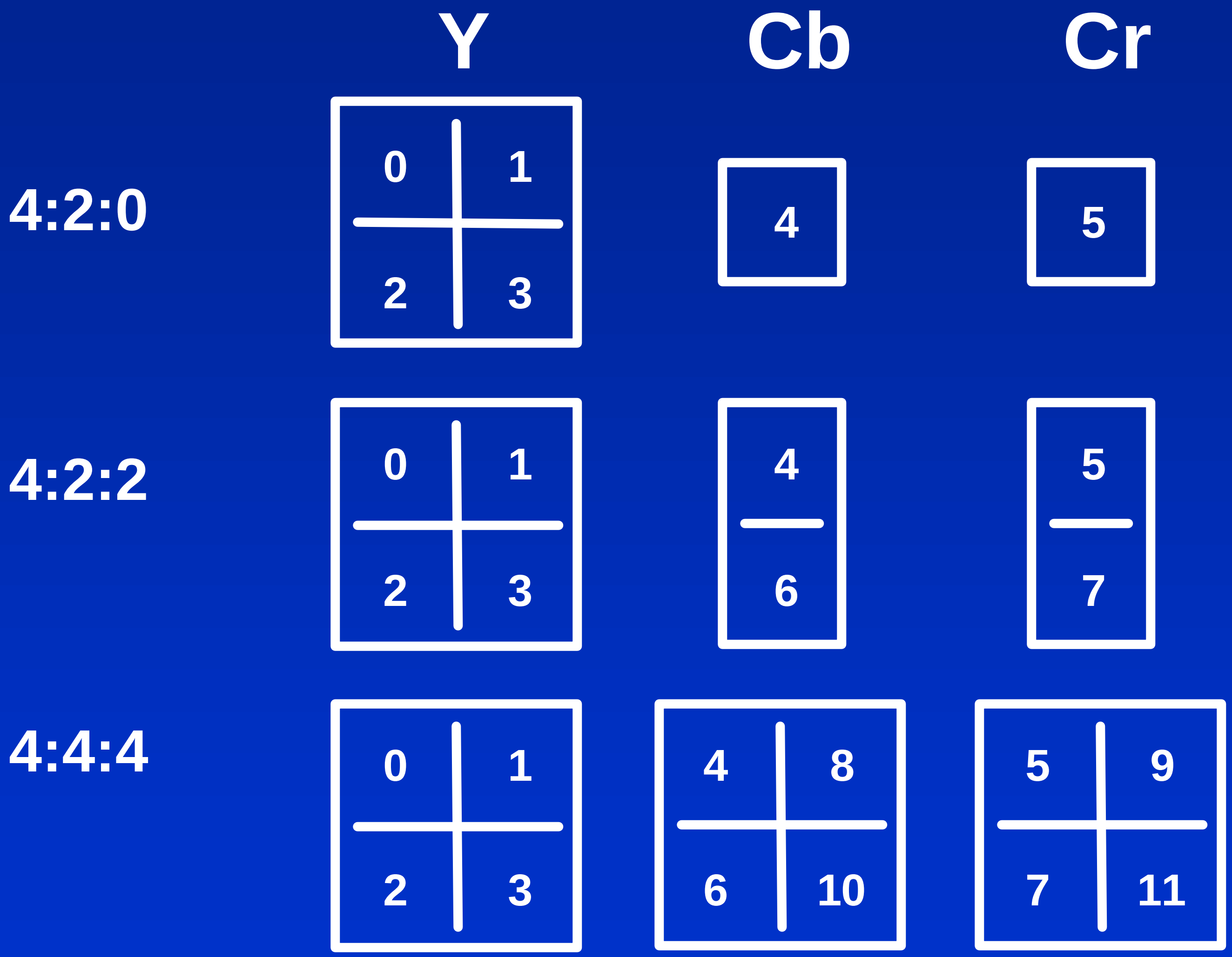


4:2:0

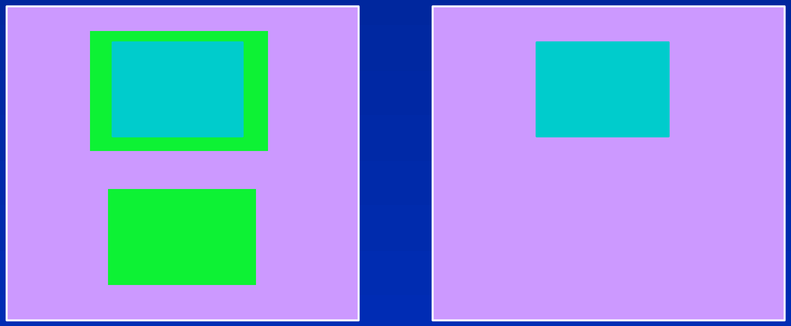
× 1 Luminance sample Y

○ 2 Chrominance samples Cb, Cr

Macroblocks & Chroma Formats

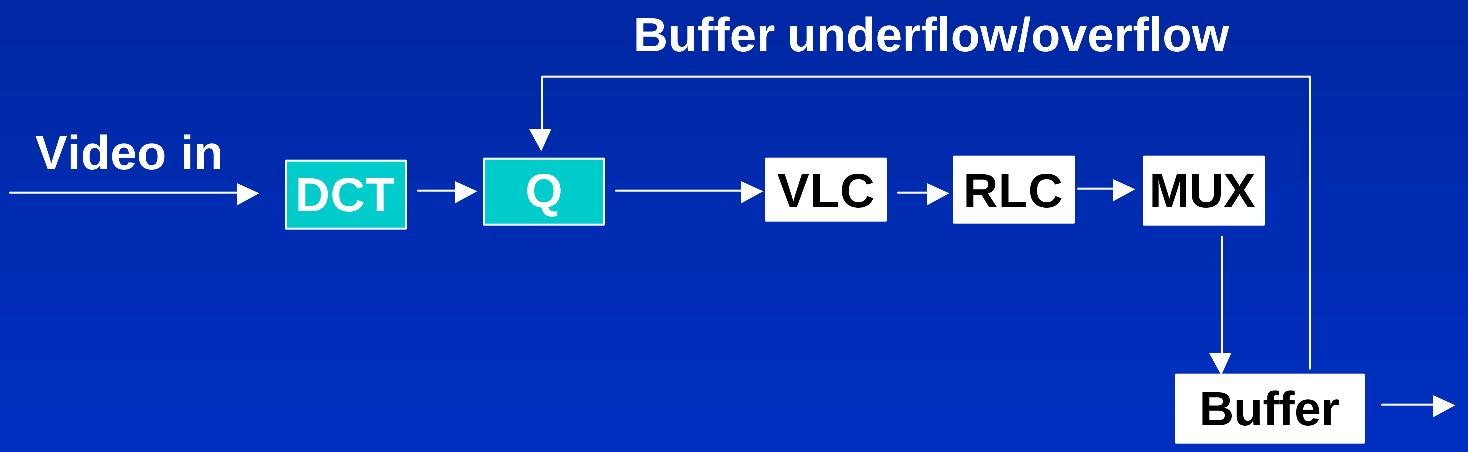


IntRA-Frame Coding

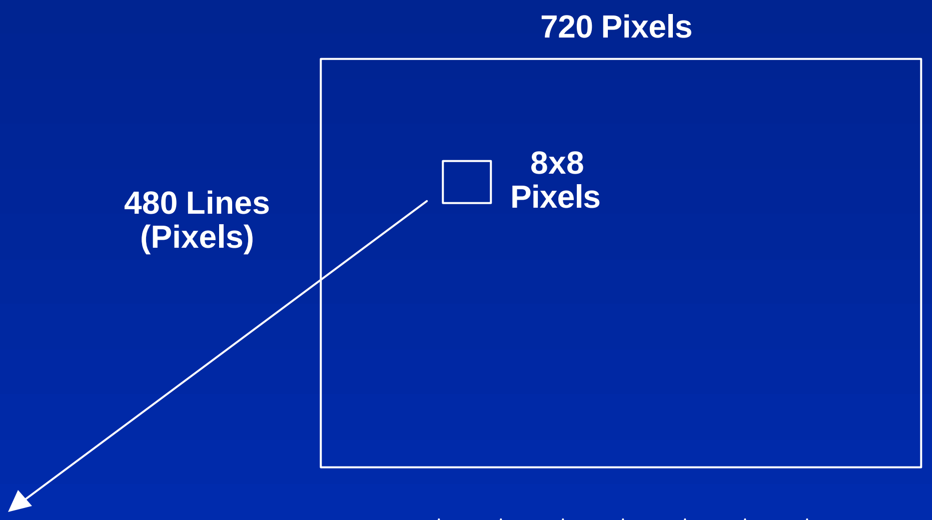
I	II	III
		
Reduce Size	Sub-sample	Transform then Quantize

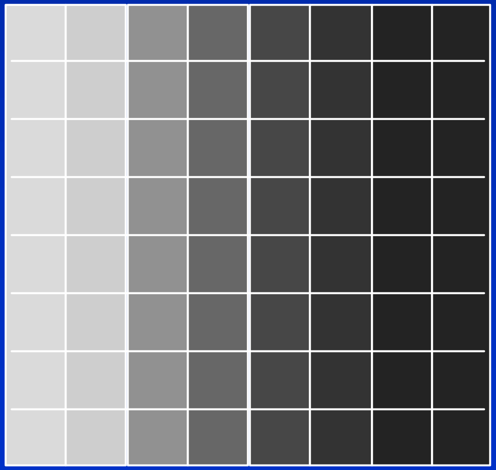
MPEG-2 Encoder

Discrete levels to frequency coefficient conversion



Discrete CosineTransform





Picture

0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5

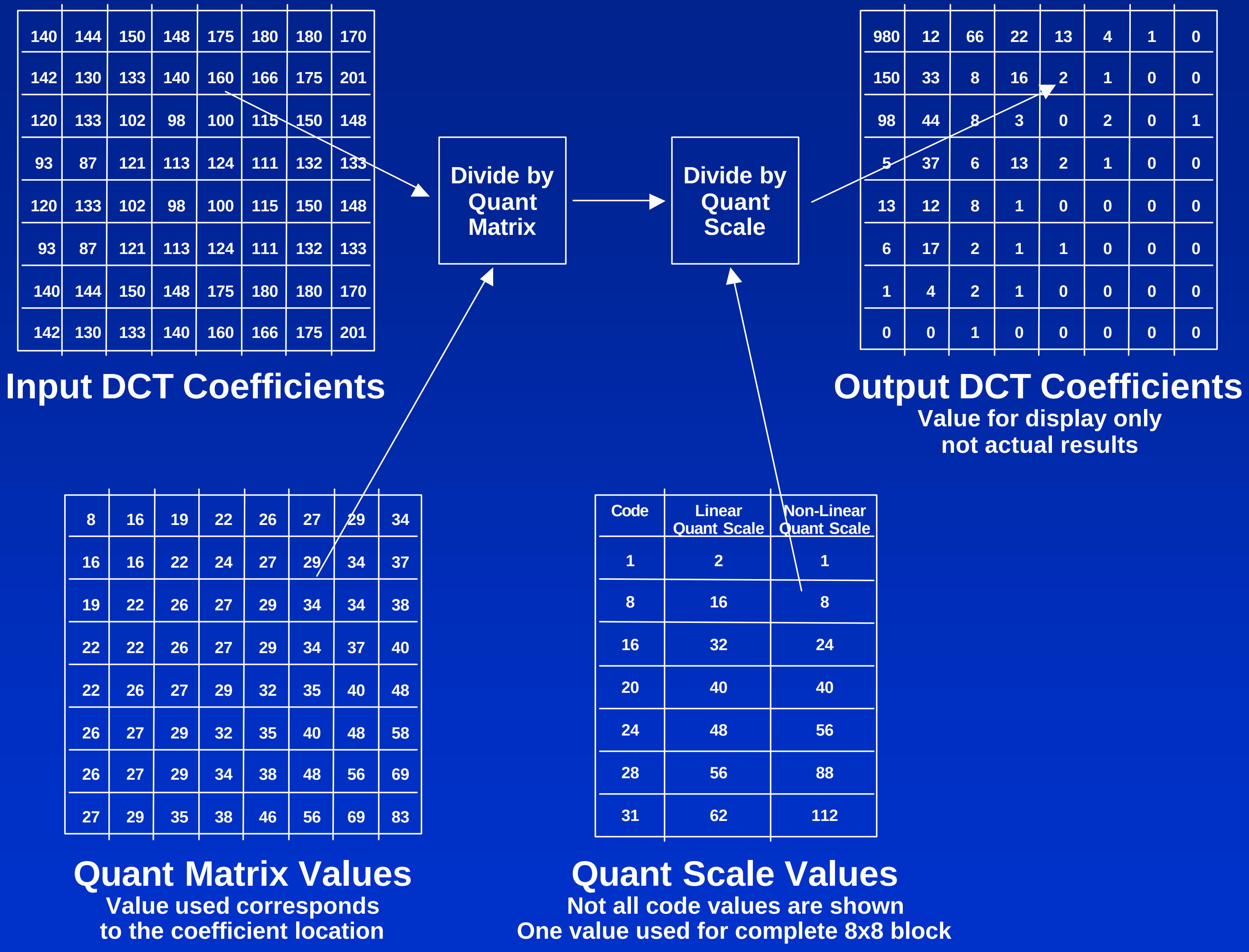
Sample Values

43.8	-40	0	-4.1	0	-1.1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

DCT Coefficients

DCT/Quantizer

- The eye is less sensitive to high frequency
- DCT matrix coefficients divided by quantization matrix
- To minimize distortion the quantization step must be small
- The quantizer is non-linear giving further bit reduction



140	144	150	148	175	180	180	170
142	130	133	140	160	166	175	201
120	133	102	98	100	115	150	148
93	87	121	113	124	111	132	133
120	133	102	98	100	115	150	148
93	87	121	113	124	111	132	133
140	144	150	148	175	180	180	170
142	130	133	140	160	166	175	201

Input DCT Coefficients

980	12	66	22	13	4	1	0
150	33	8	16	2	1	0	0
98	44	8	3	0	2	0	1
5	37	6	13	2	1	0	0
13	12	8	1	0	0	0	0
6	17	2	1	1	0	0	0
1	4	2	1	0	0	0	0
0	0	1	0	0	0	0	0

Output DCT Coefficients

Value for display only
not actual results

8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	48	56	69
27	29	35	38	46	56	69	83

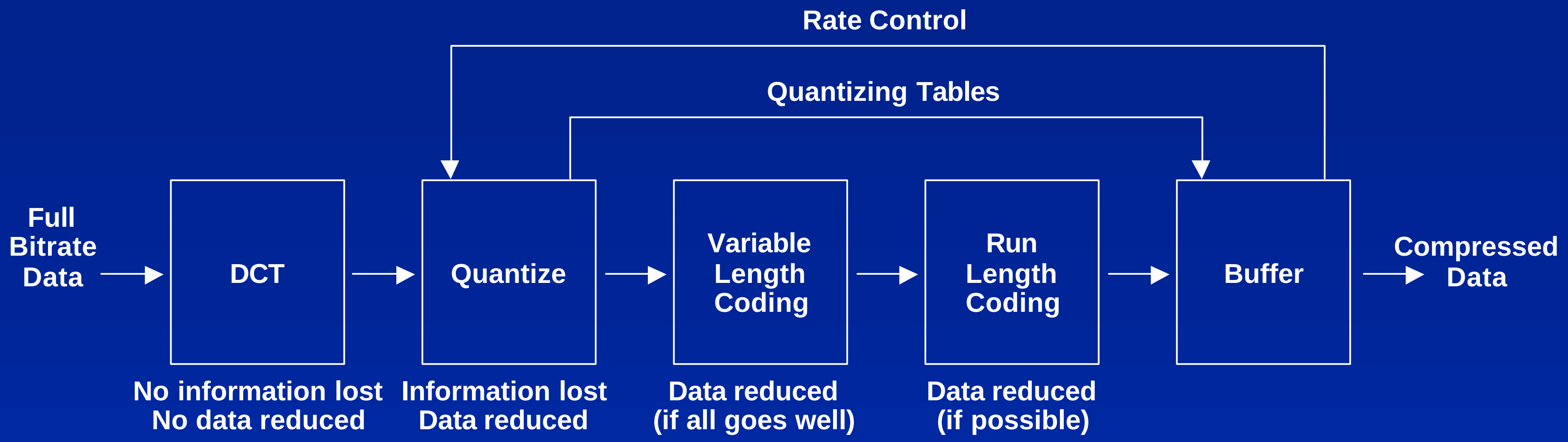
Quant Matrix Values

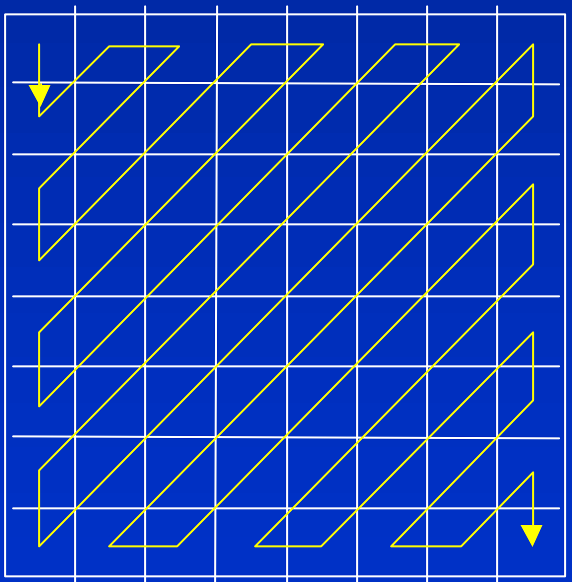
Value used corresponds
to the coefficient location

Code	Linear Quant Scale	Non-Linear Quant Scale
1	2	1
8	16	8
16	32	24
20	40	40
24	48	56
28	56	88
31	62	112

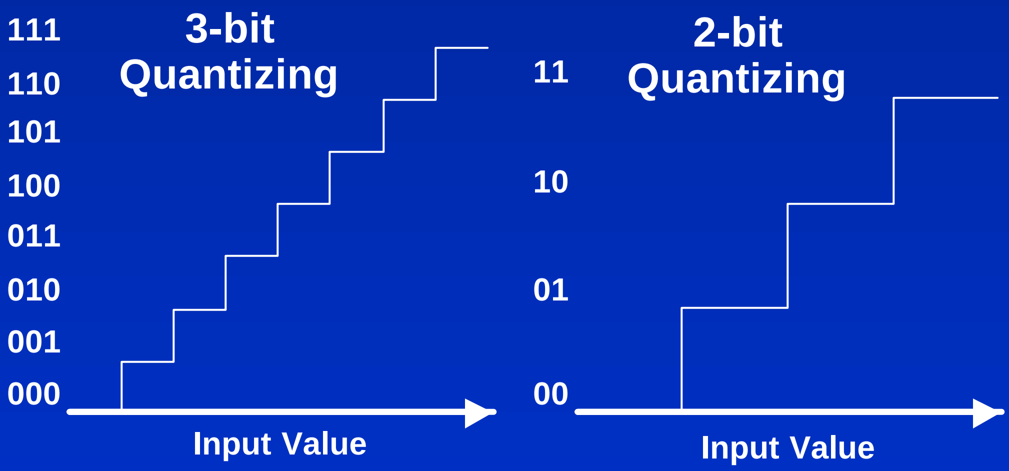
Quant Scale Values

Not all code values are shown
One value used for complete 8x8 block





Coefficient processing order
to encourage runs of 0s



Quantizing

Reduce the number of bits for each coefficient.
Give preference to certain coefficients.
Reduction can differ for each coefficient.

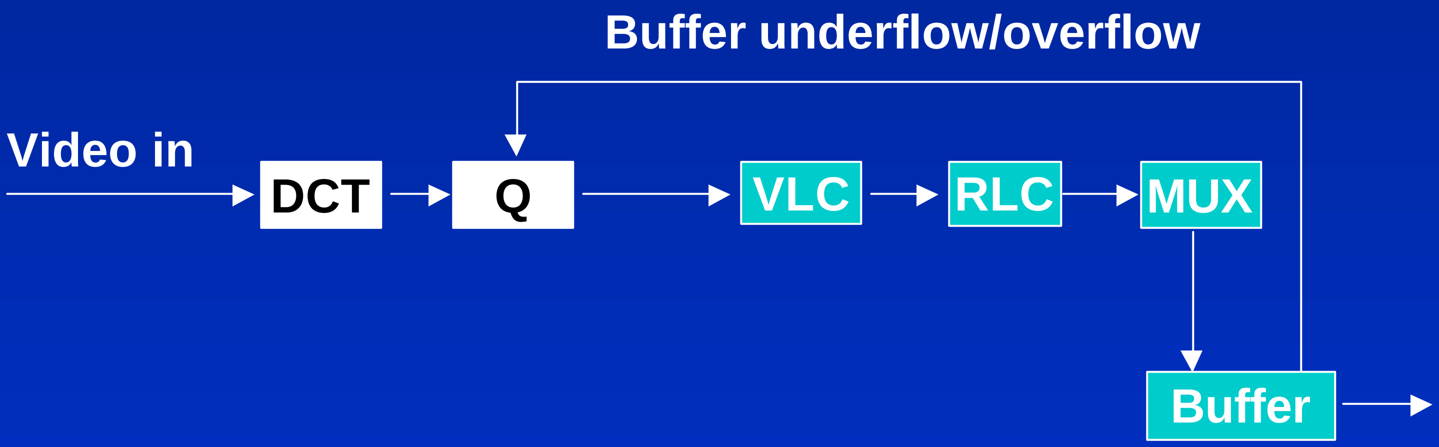
Variable Length Coding

Use short words for
most frequent values
(like Morse Code)

Run Length Coding

Send a unique code
word instead of strings
of zeros

MPEG-2 Encoder Continued



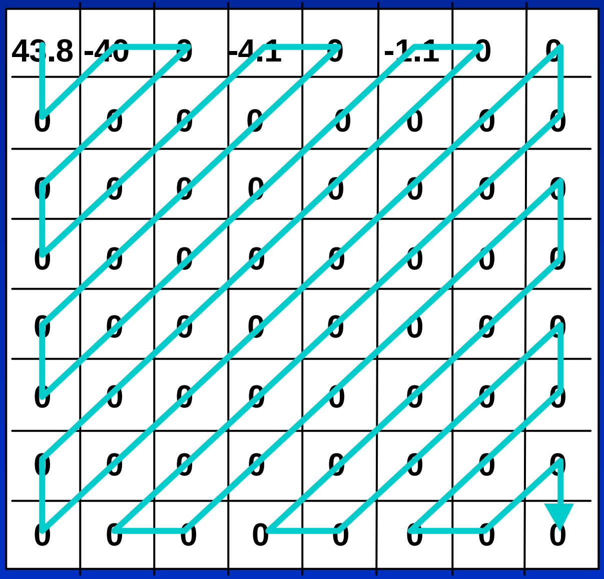
Entropy Coding

- Use short code words for the most common symbols
- Huffman codes

Symbol	Probability	Code Word
A	0.5	0
B	0.25	10
C	0.125	110
D	0.0625	1110
E	0.03125	11110
F	0.03125	11111

Run Length Coding (RLC)

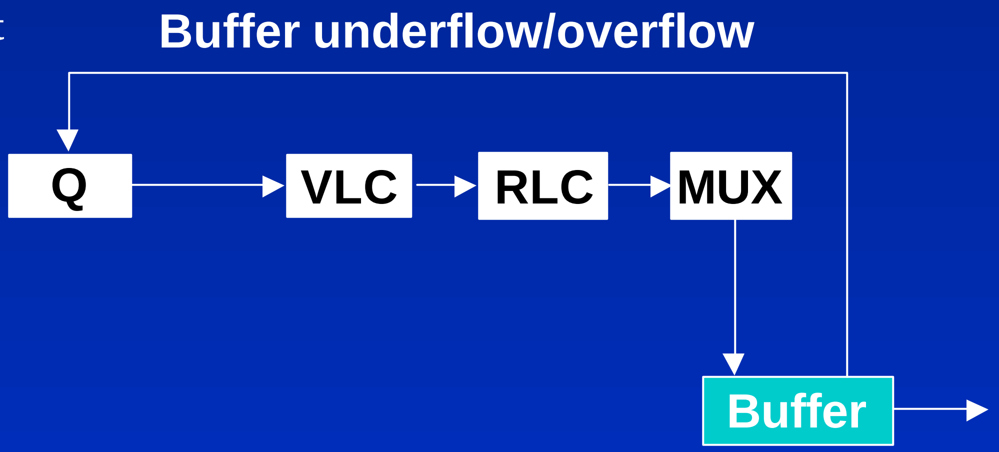
- The effect of the various coding techniques up to this point is to reduce most of the values to zero or near zero.
- The output of the VLC has strings of zeros.
- This can be optimized by the pattern used to read the data.
- The string of zeros is replaced with a code representing the *n* of zeros in the string.



43.8	-40	0	-4.1	0	-1.1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

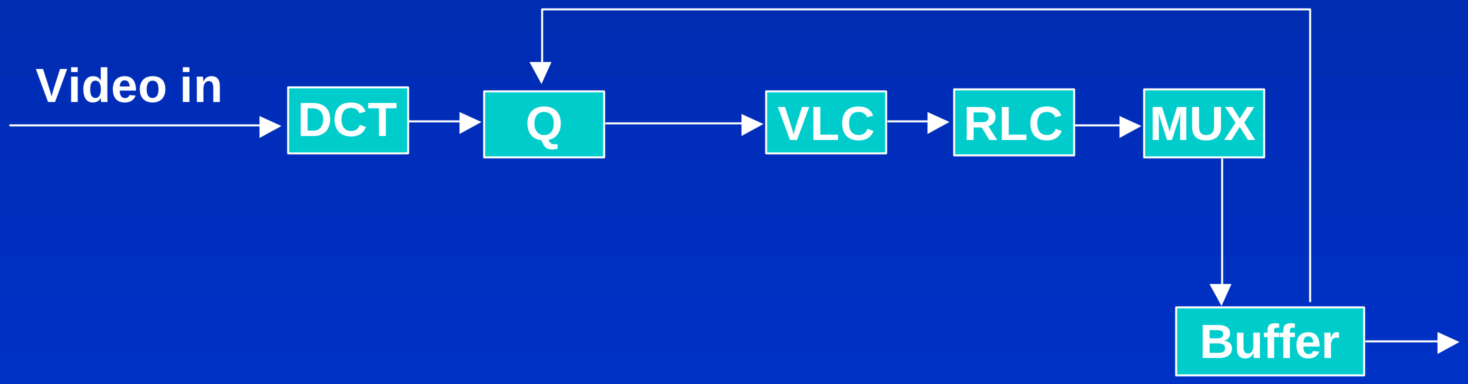
Buffer

- The Buffer maintains a constant output data rate.
- If it begins to get empty it sends underflow to the quantizer to increase the
  bit rate output.
- If it begins to get full it sends overflow back to the quantizer to reduce its bit rate output.

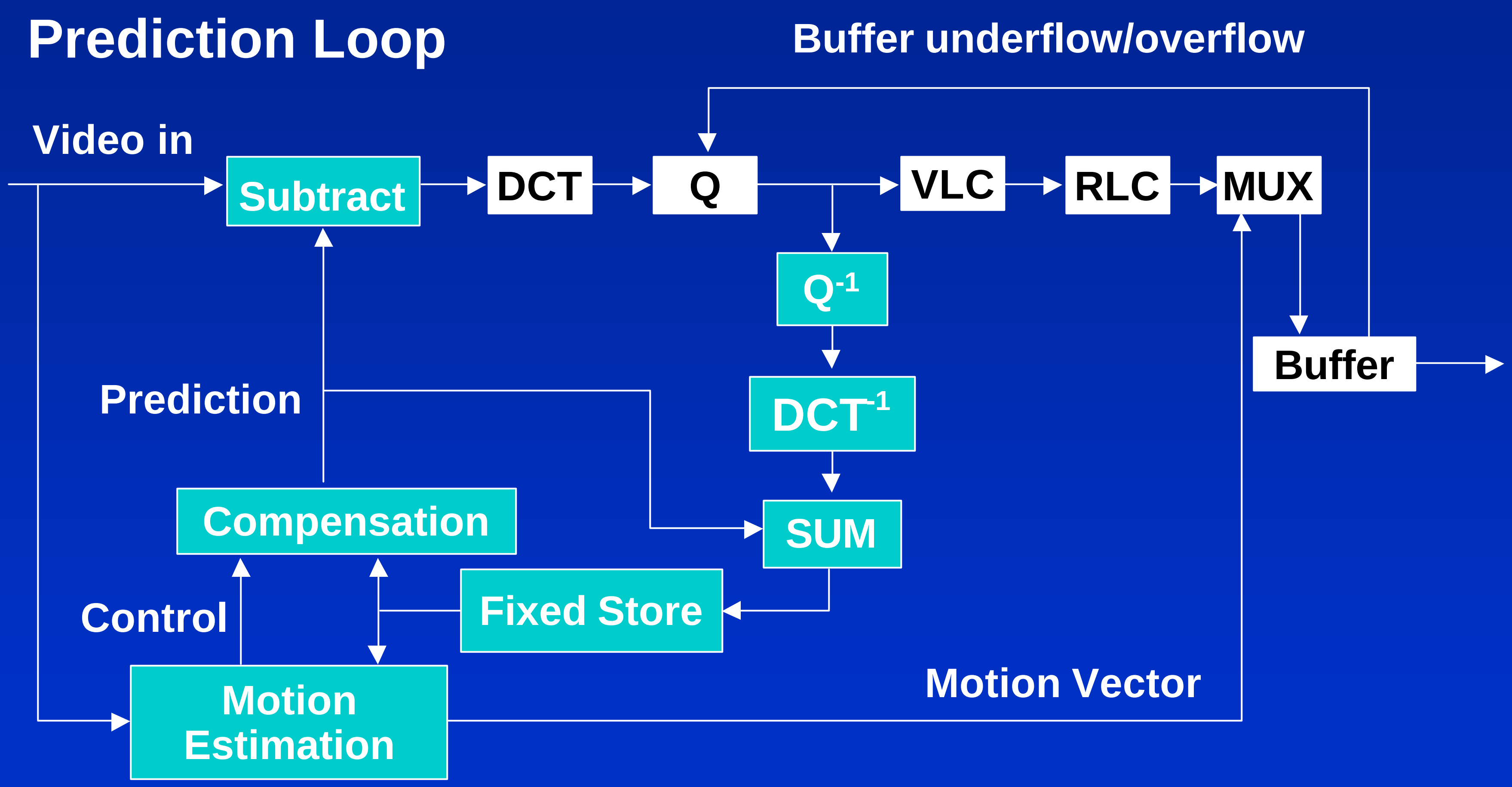


Compression without Motion

- At this point the encoder is basically JPEG or MPEG-1
- MPEG-1 does use motion compensation but in progressive frame

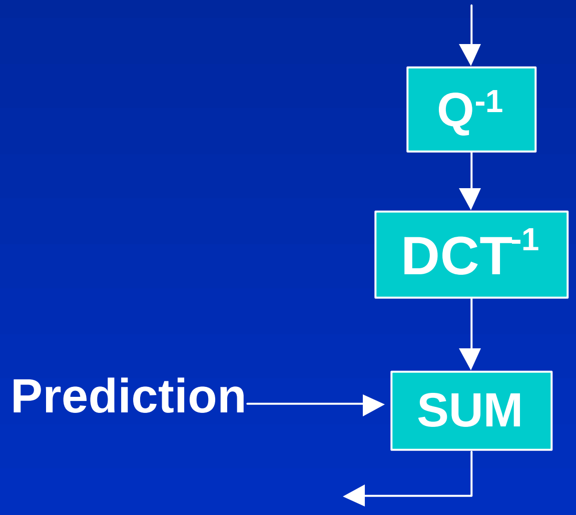


Inter-Frame Coding (Motion Compensation)



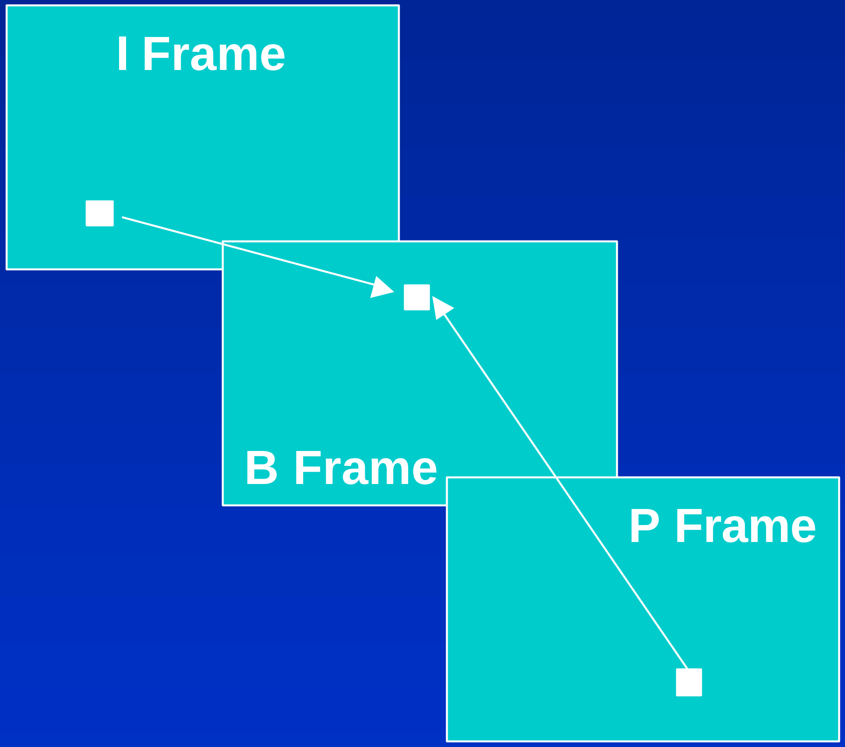
Inverse Quantizer, DCT and Sum

- The function of this area is to return the data to input format.
- The data is then summed with the prediction data.
- The data is now a representation of actual data and predicted data.
- The data can now be checked for errors in prediction and a new prediction made.



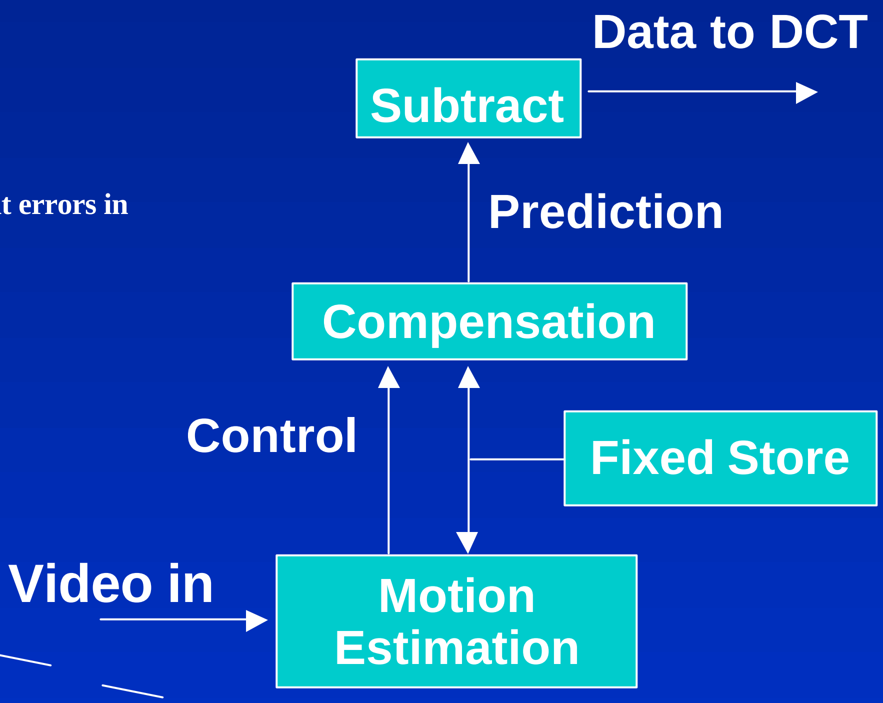
Macroblock Search

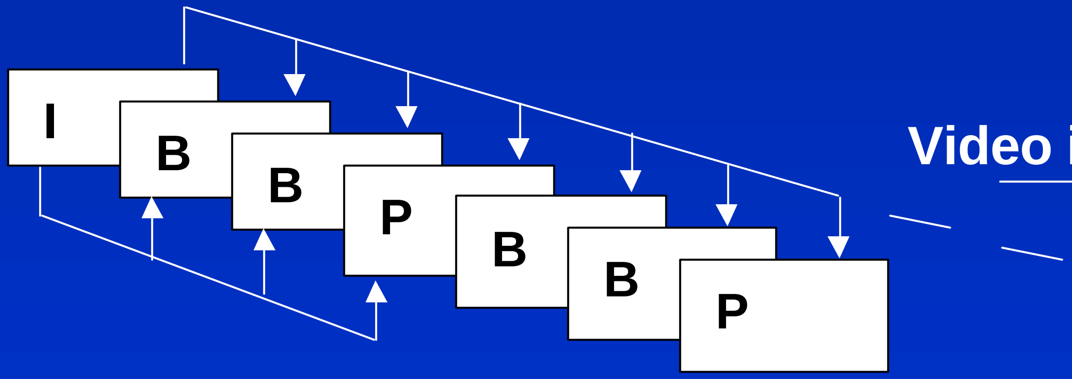
- The macroblock is checked against previous frames for
  a match.
- If a match is found only the motion vector need be encoded.
- The Encoders search within
  1/2 pixel accuracy.
- This results in lossless compression with a large reduction in bit rate.
- Checks are forward and back-ward from the anchor frames.



Motion Compensation (Inter-Frame Coding)

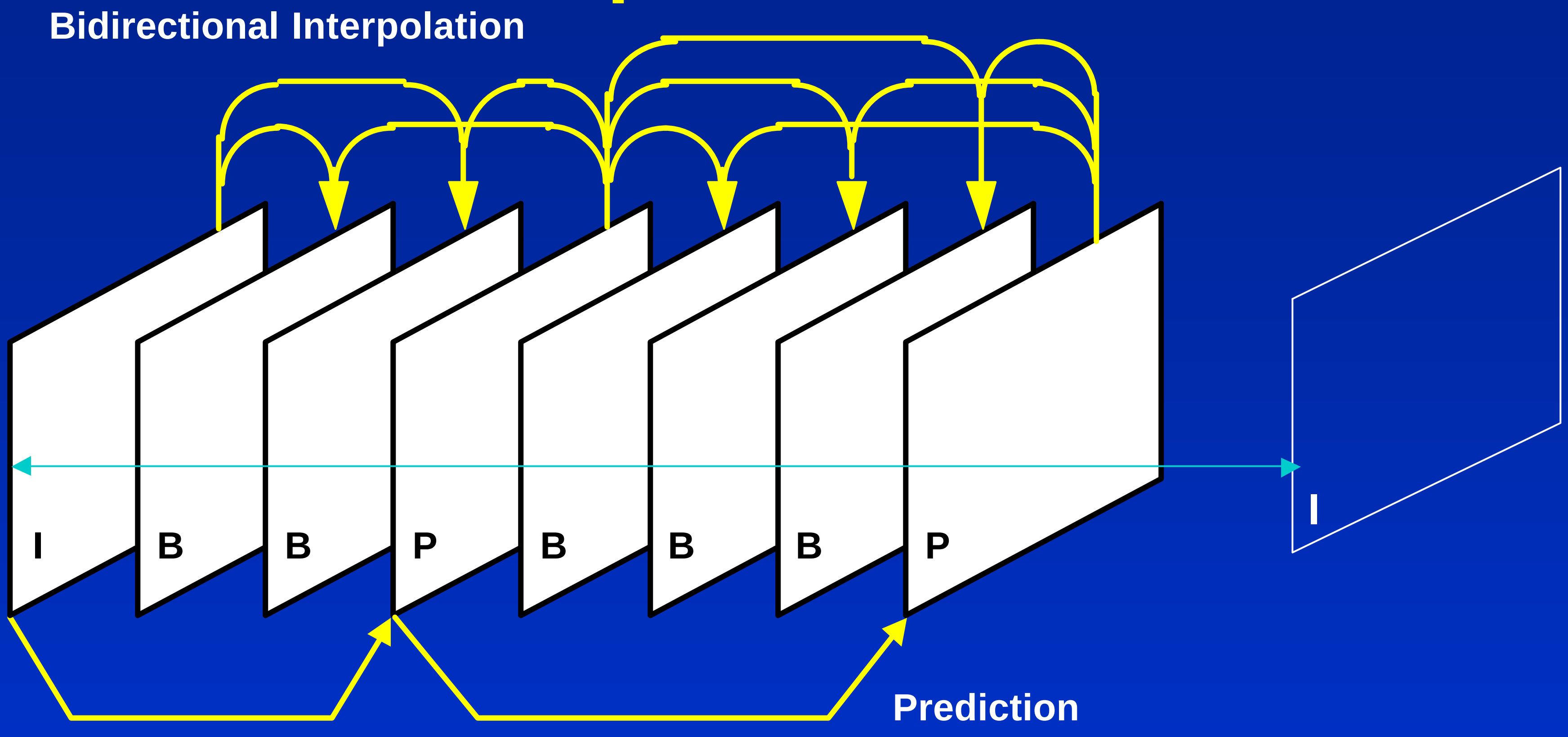
- I frames are stored uncompressed
- P frames are predicted from I frames
- B frames are generated forward and backward
- I and P frames are transmitted anchors
- In practice, a new I frame is stored every 13 to 16 frames to prevent errors in prediction from lasting too long





I and P frames are called anchor frames

Picture Types & Groups of Pictures



I pictures: Composed of intra macroblocks only
P pictures: Contain forward motion compensation and intra macroblocks
B pictures: Contain forward, backward & bi-directional MC plus intra macroblocks

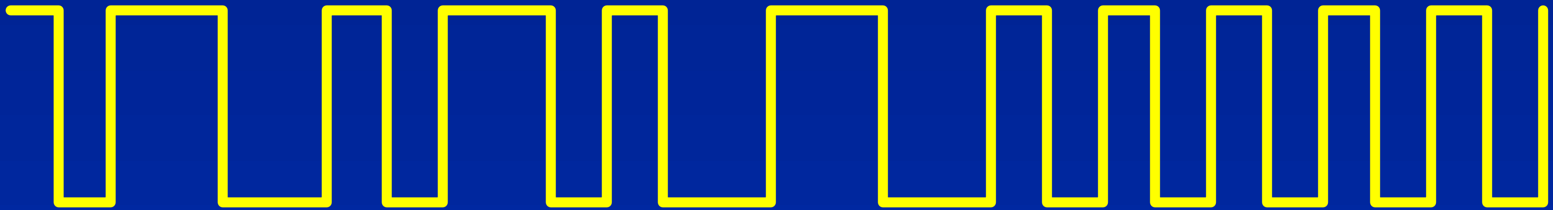
MPEG-2 Profiles and Levels

LEVEL / PROFILE	SIMPLE	MAIN		SNR	SPATIAL	HIGH
HIGH		4:2:0 1920 x 1152 80 Mbps I, P, B				4:2:0, 4:2:2 1920 x 1152 100 Mbps I, P, B
HIGH-1440		4:2:0 1440 x 1152 60 Mbps I, P, B			4:2:0 1440 x 1152 60 Mbps I, P, B	4:2:0, 4:2:2 1440 x 1152 80 Mbps I, P, B
MAIN	4:2:0 720 x 576 15 Mbps I, P	4:2:0 720 x 576 15 Mbps I, P, B	4:2:2 720 x 608 50 Mbps I, P, B	4:2:0 720 x 576 15 Mbps I, P, B		4:2:0, 4:2:2 720 x 576 20 Mbps I, P, B
LOW		4:2:0 352 x 288 4 Mbps I, P, B		4:2:0 352 x 288 4 Mbps I, P, B		

MPEG Audio

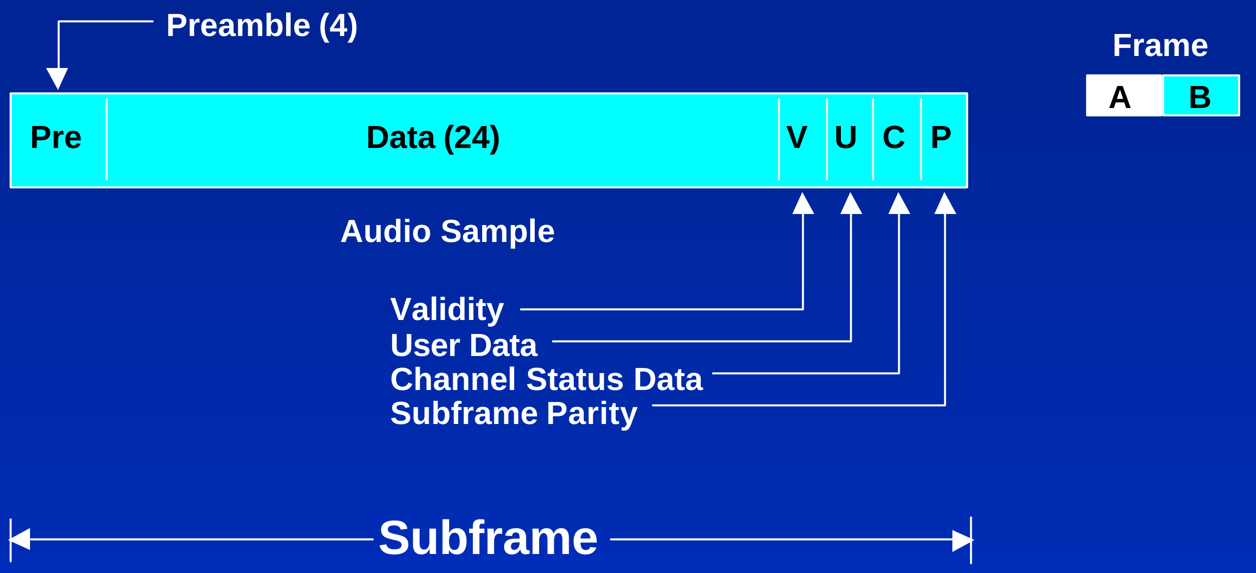
- AES/EBU Audio
- Compression Techniques
- Data structures

AES Digital Audio Signal



- Type of Manchester code - bi-phase mark coding
- Basic characteristics
- No dc content
- Information contained in transitions
- Immune to polarity reversal
- Clock signal embedded

AES/EBU Frames & Subframes



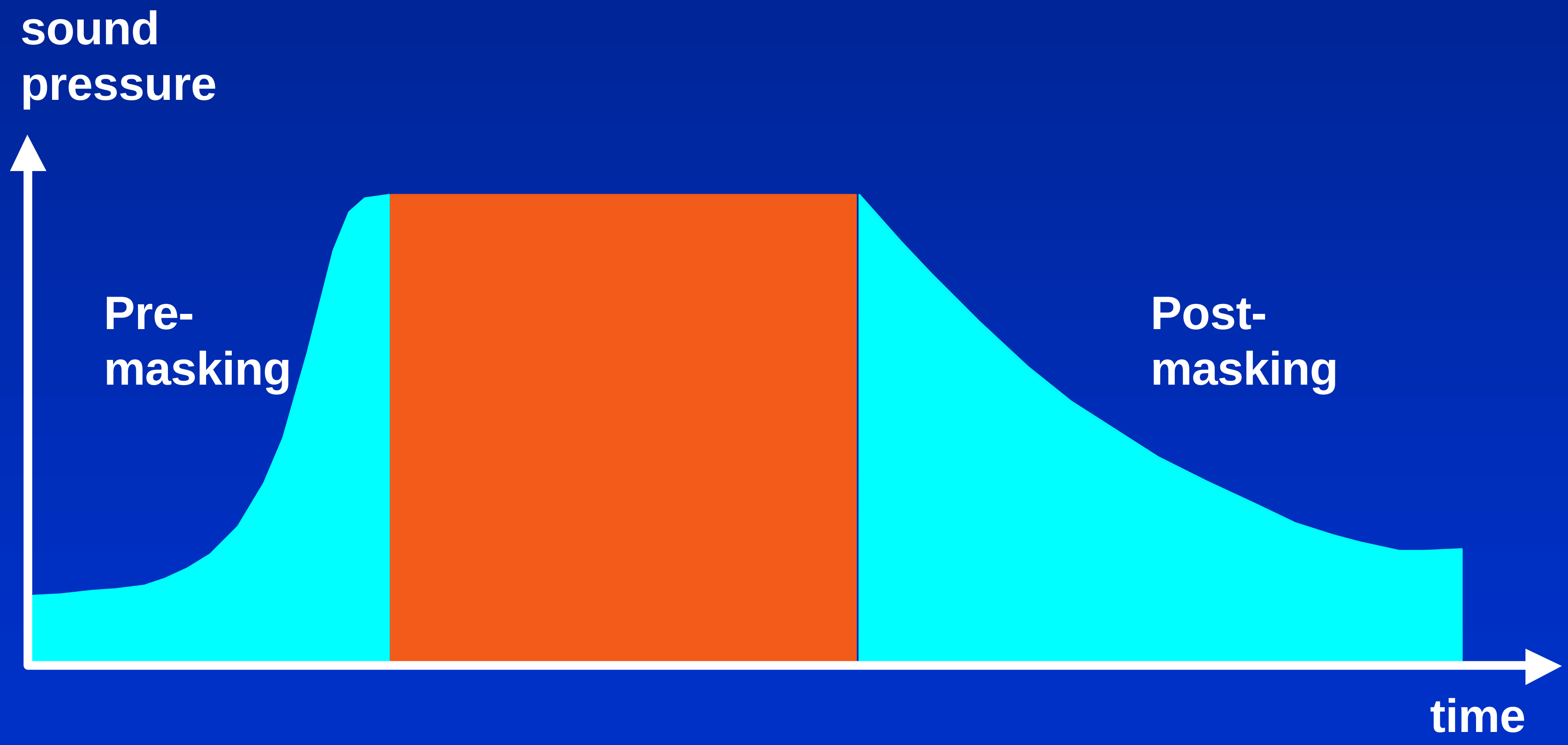
- 32 Bits/subframe
- 4 Preamble Bits, 24 Data Bits, 1 Validity,
- 1 User, 1Channel Status & 1 Parity

Psychoacoustic Models

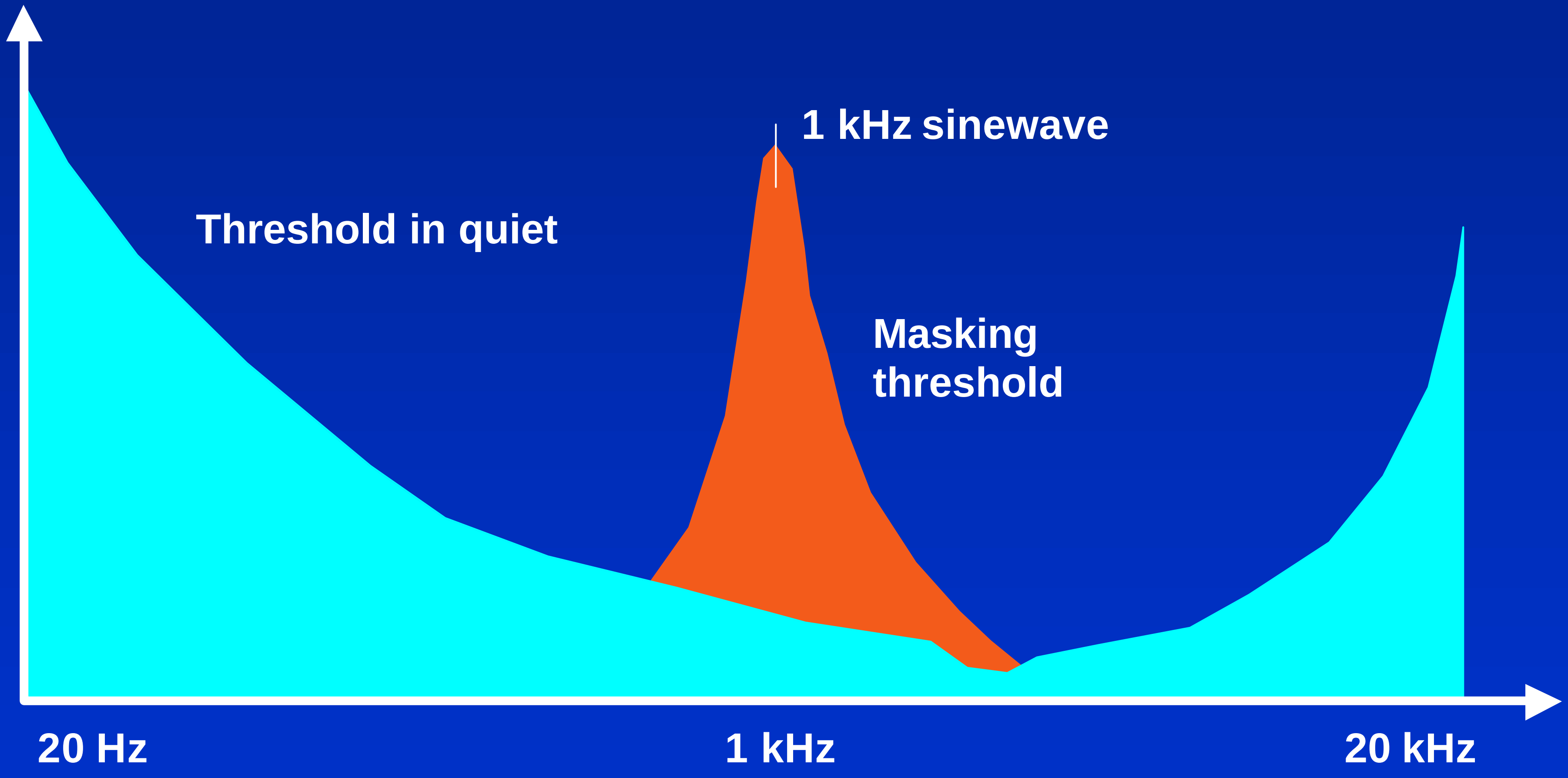
- Pre-Masking
- Post-Masking
- Simultaneous Masking

Premise:
If you can’t hear it, don’t send it.

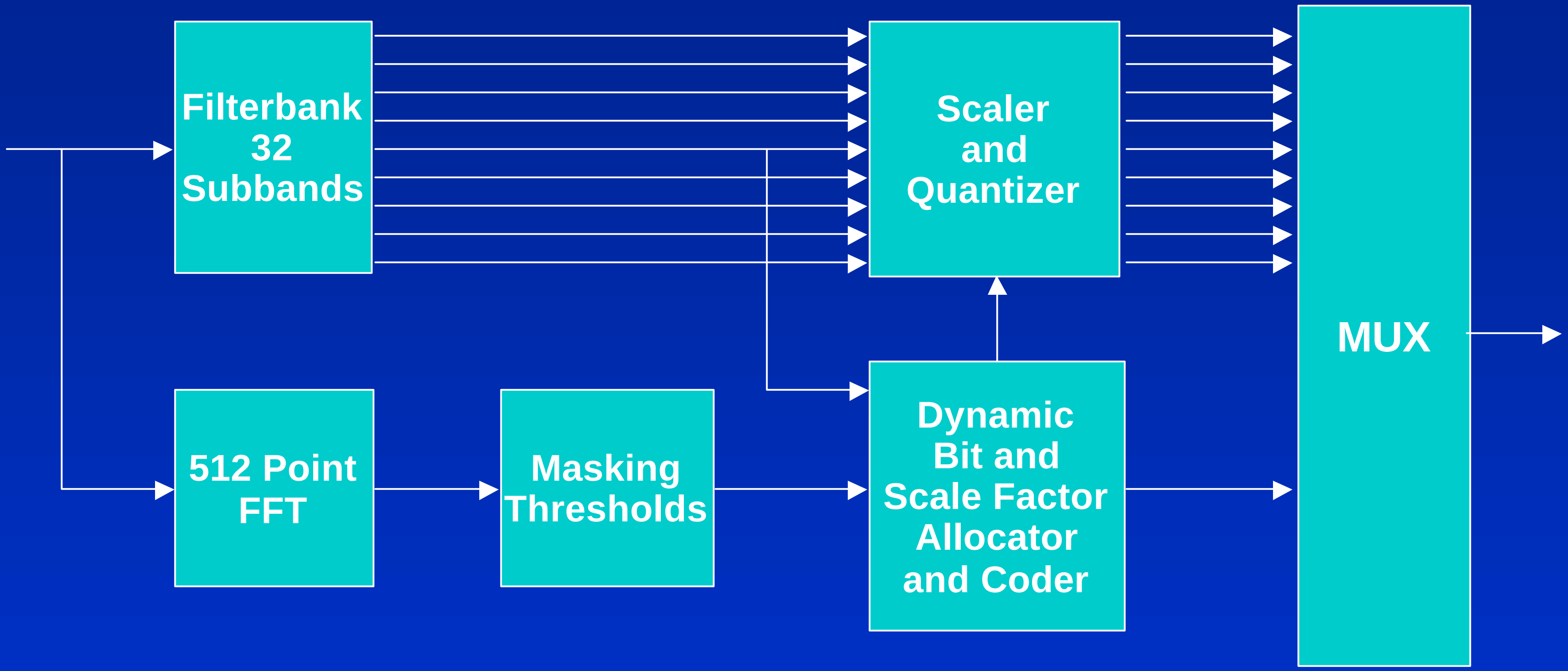
Temporal Masking



Simultaneous Masking



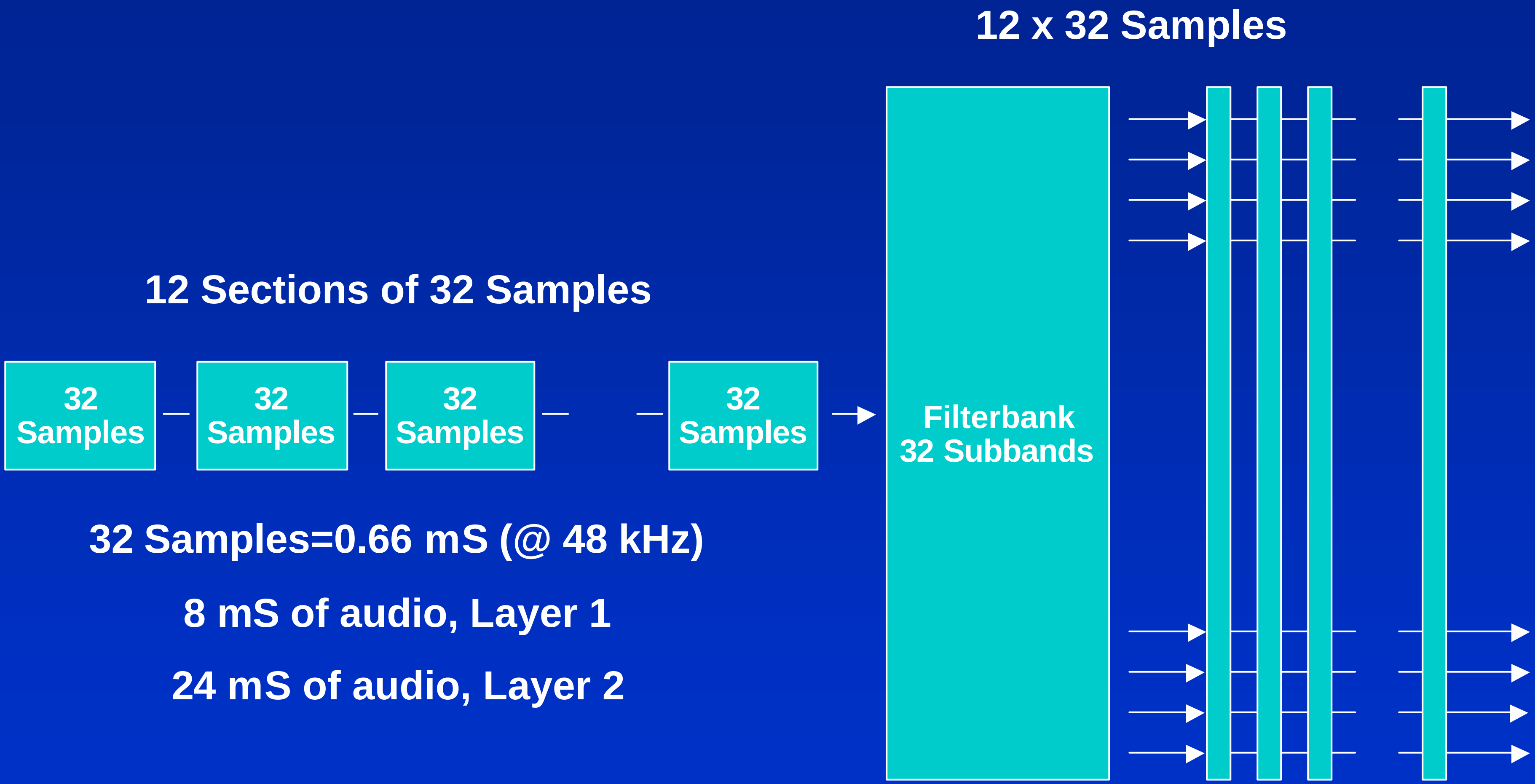
MPEG Audio Encoder



Audio Frame input PCM samples
384 for Layer 1
3 * 384 = 1152 for Layer 2

Audio Time Frame

12 Sections of 32 Samples

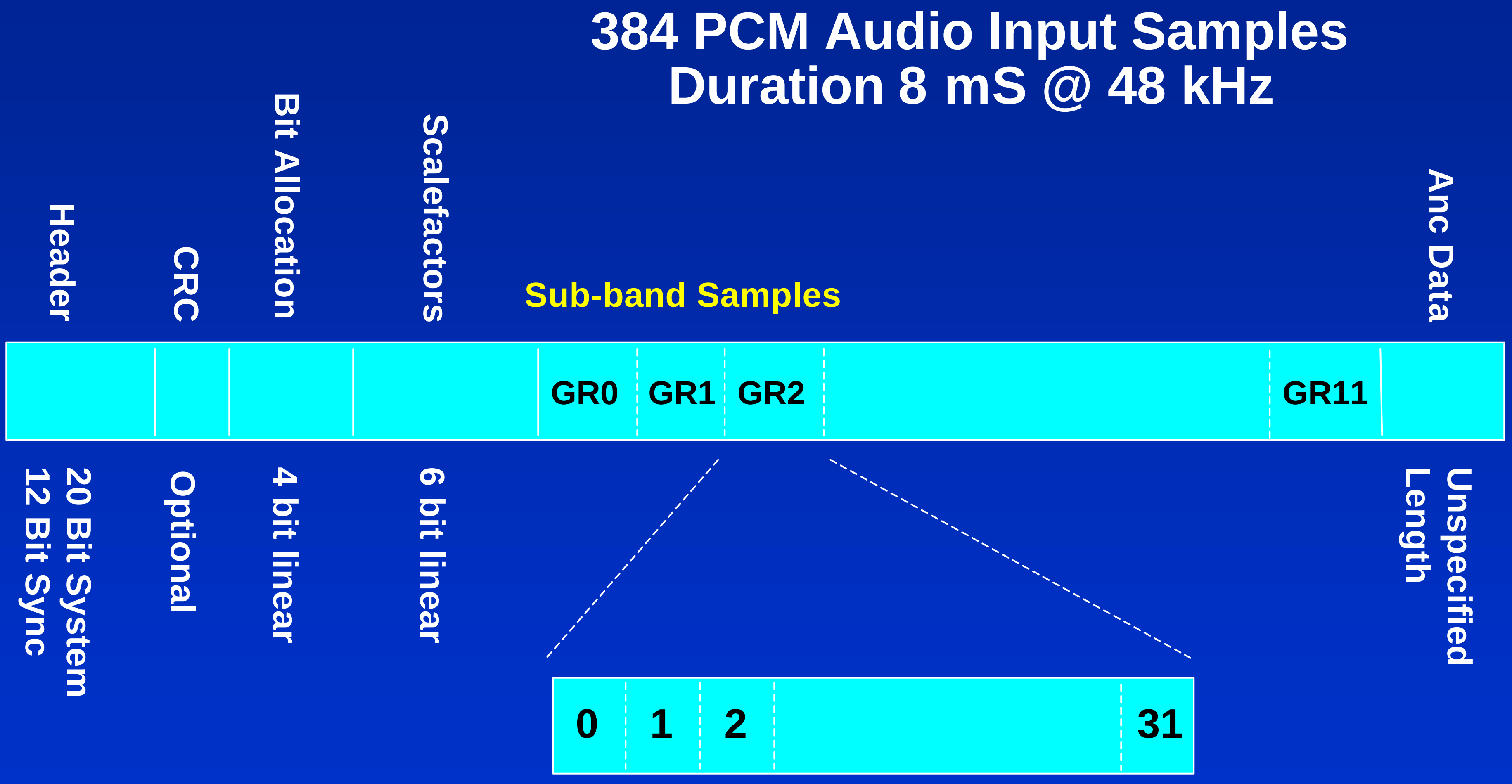


32 Samples=0.66 mS (@ 48 kHz)

8 mS of audio, Layer 1

24 mS of audio, Layer 2

Layer I Frame Structure

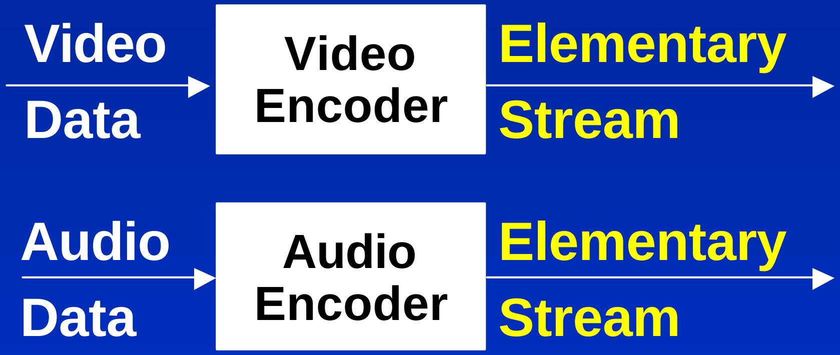


MPEG-2

- MPEG-2 ISO/IEC 13818/Recommendation H.222.0 (1994)
  - 13818-1 system level coding (audio and video multiplex)
  - 13818-2 coding and decoding of video data
  - 13818-3 coding of audio data
  - 13818-4 compliance testing of the other three
  - Standard covers the decoder methodology and transport stream syntax

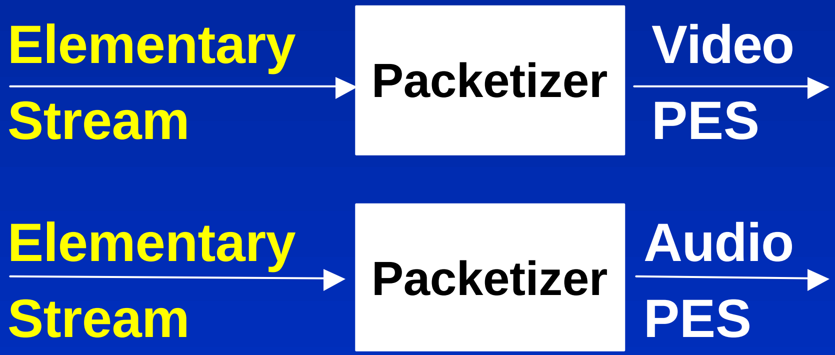
MPEG-2 Encoder/Compressor

- MPEG-2 requires component digital video (Rec. 601).
- The MPEG-2 encoder/ compression product *can* include composite decoders and/or A to D converters.
- The audio input must be AES/EBU digital or the
  encoder must include a
  converter/formatter.
- Elementary Streams can be of any length.



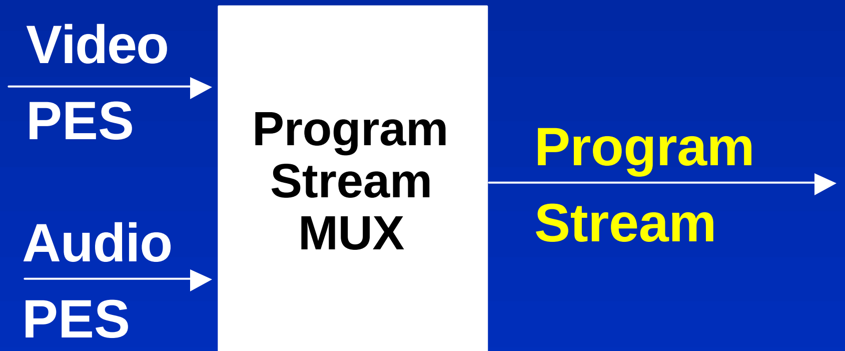
MPEG-2 Packetizer

- The Packetizer forms the Video and/or Audio into Packetized Elementary Stream (PES) packets.
- PES Packets contain:
  - Header data
  - Elementary stream data



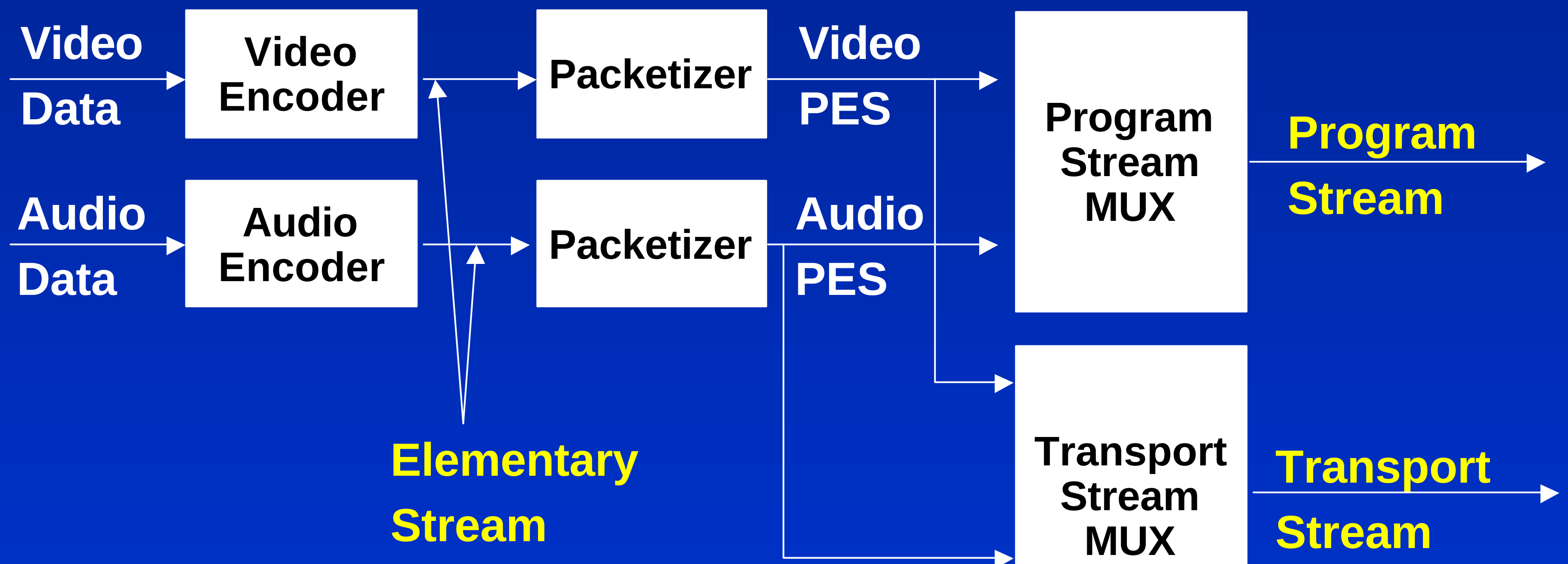
Program Stream Mux

- Combines Video and Audio PES packets into a stream.
- Designed for transmission in error and noise free environments.



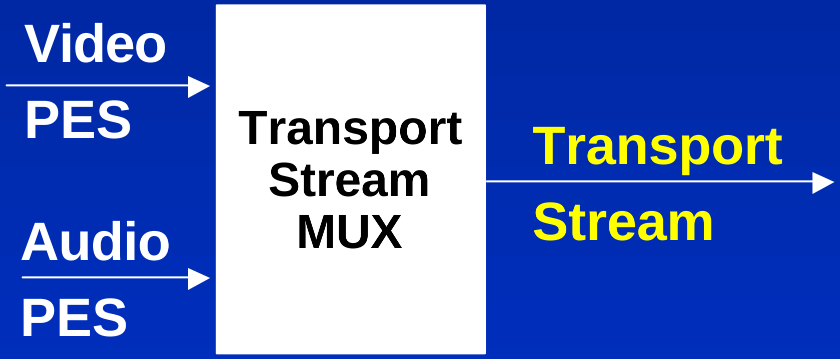
MPEG-2 Compression Data

Basic MPEG-2 data layers and terminology

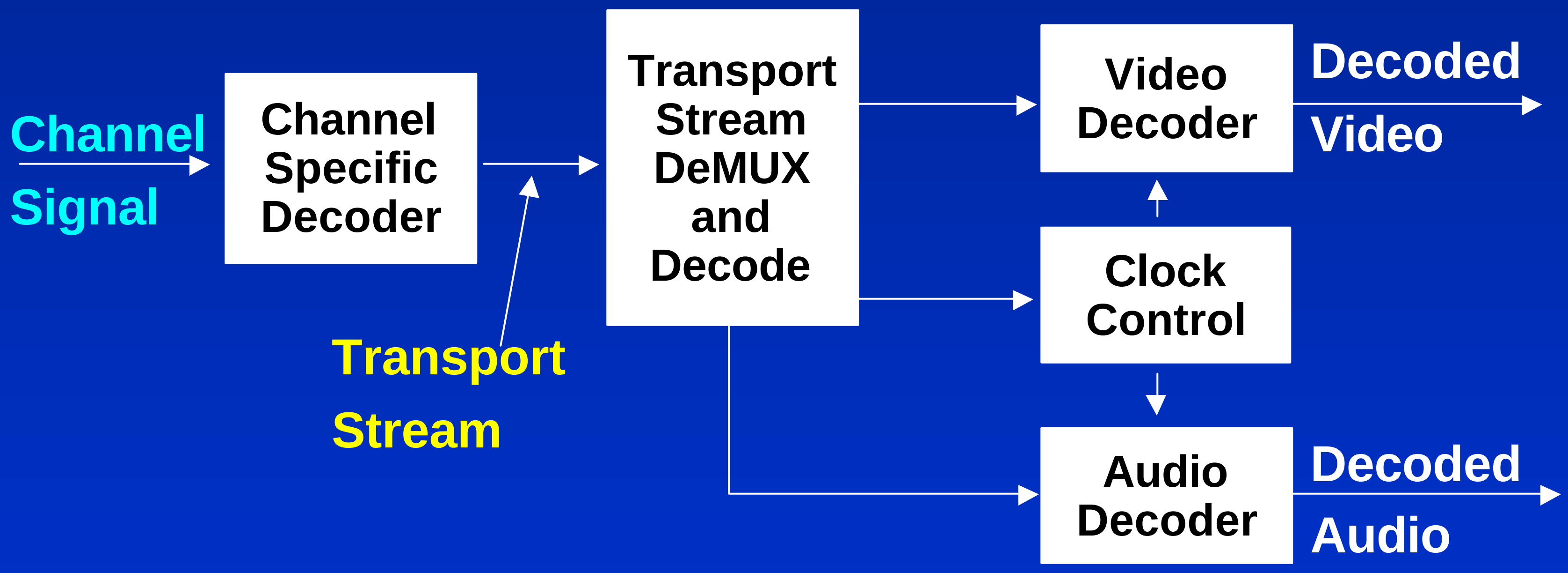


Transport Stream MUX

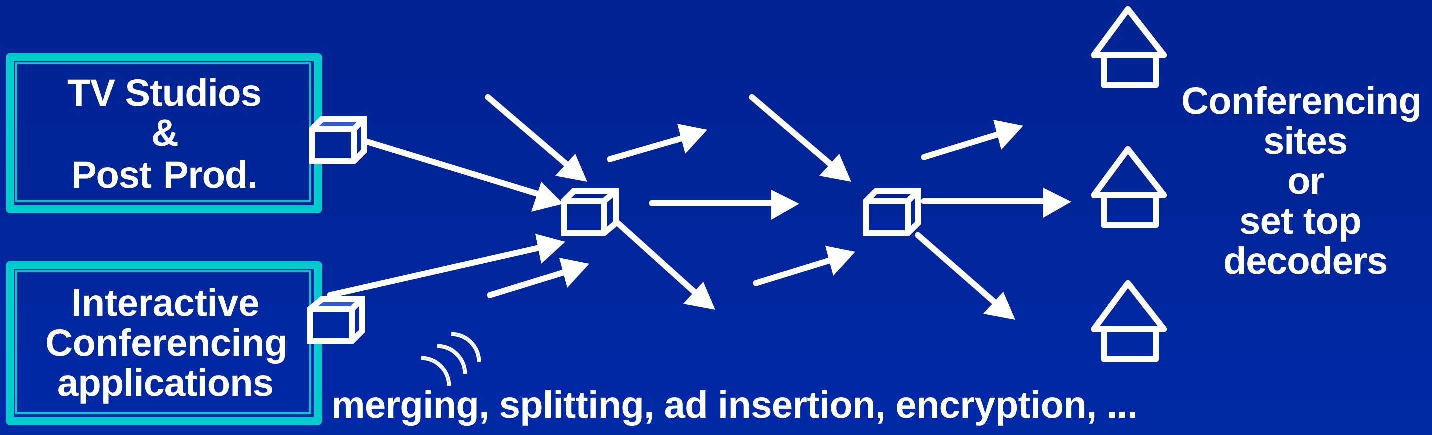
- Combines Video and Audio PES packets into a stream.
- Designed for transmission to ITU-T Rec. H.262, ISO/IEC 13818 standards, and anywhere significant errors may occur.
- The TS contains no error protection itself.
- The use of small packets provides some resistance to the results of errors.
- Combines asynchronous signals to synchronous data stream.

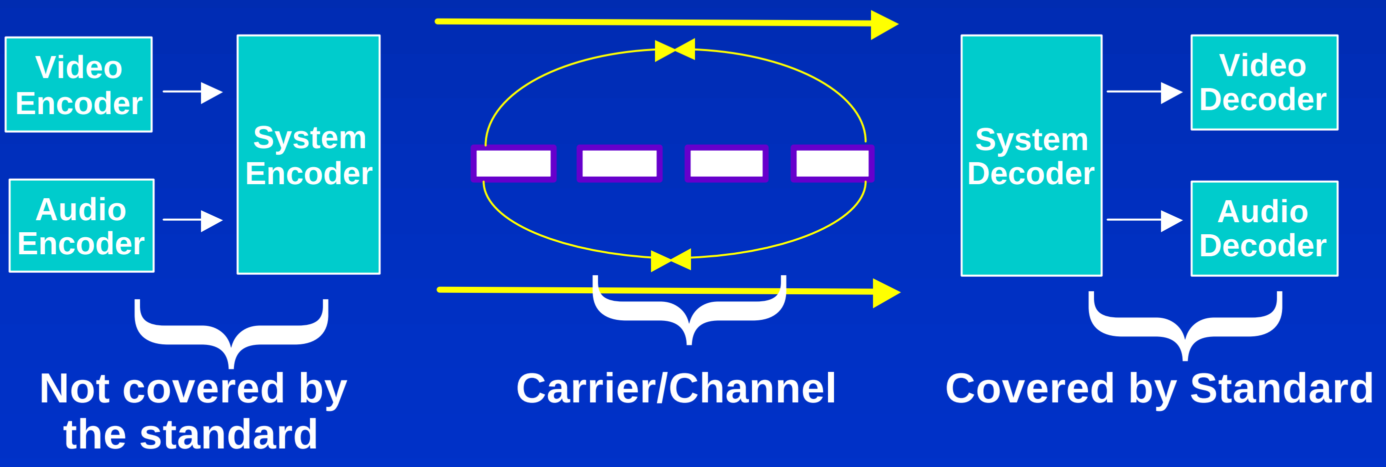


MPEG-2 Decoder

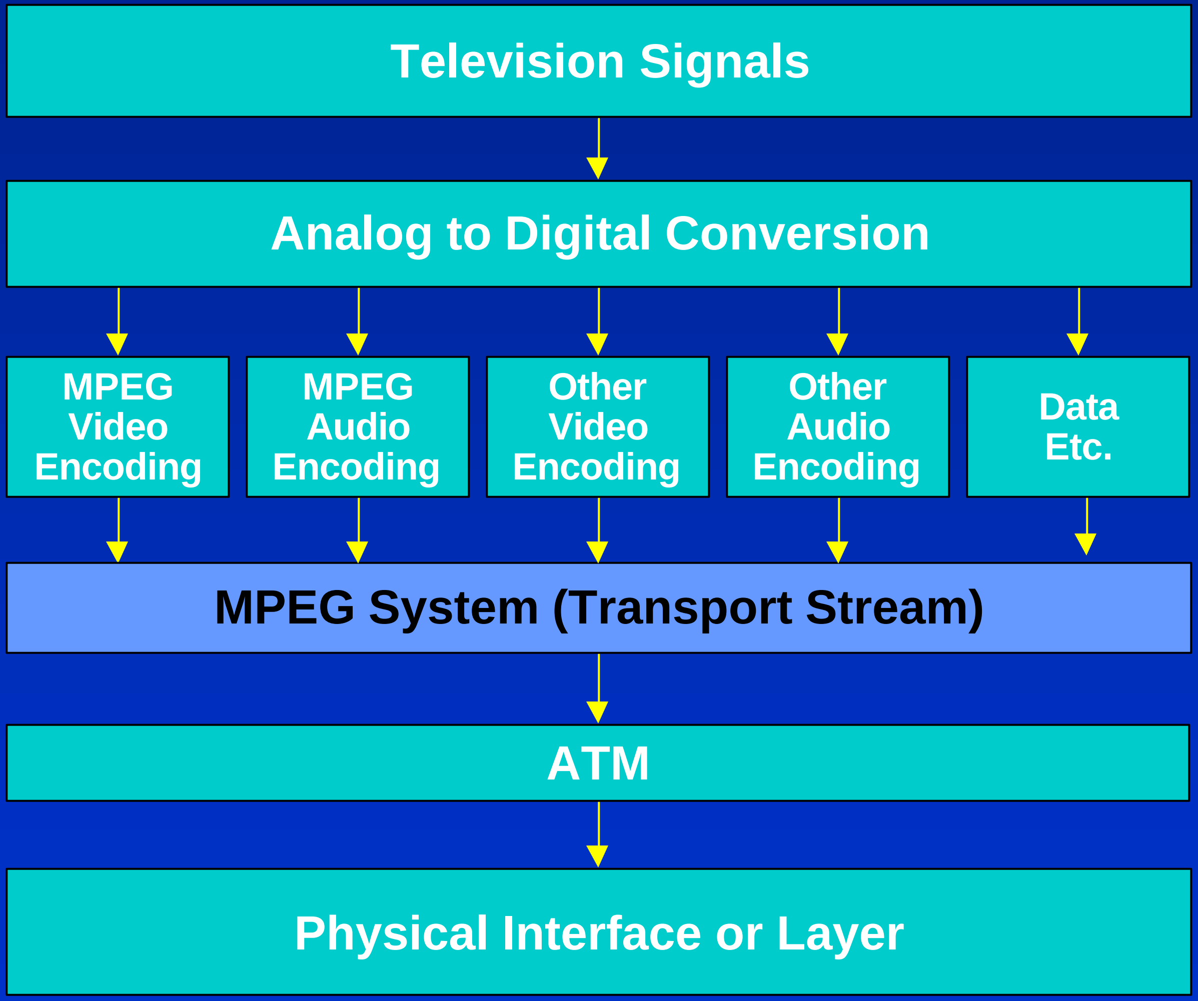


The MPEG-2 System

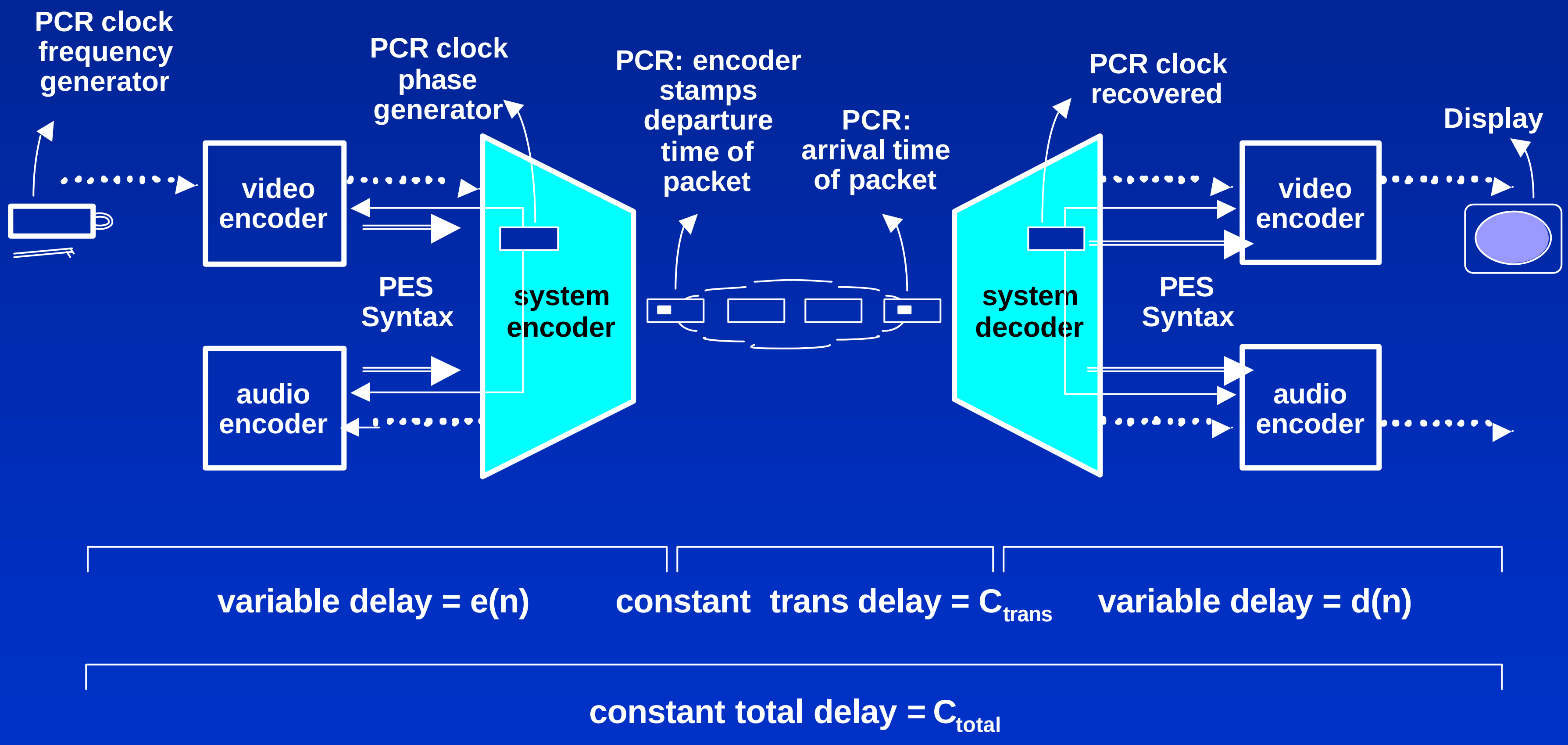

TV Studios & Post Prod.
Interactive Conferencing applications
Conferencing sites or set top decoders
merging, splitting, ad insertion, encryption, ...



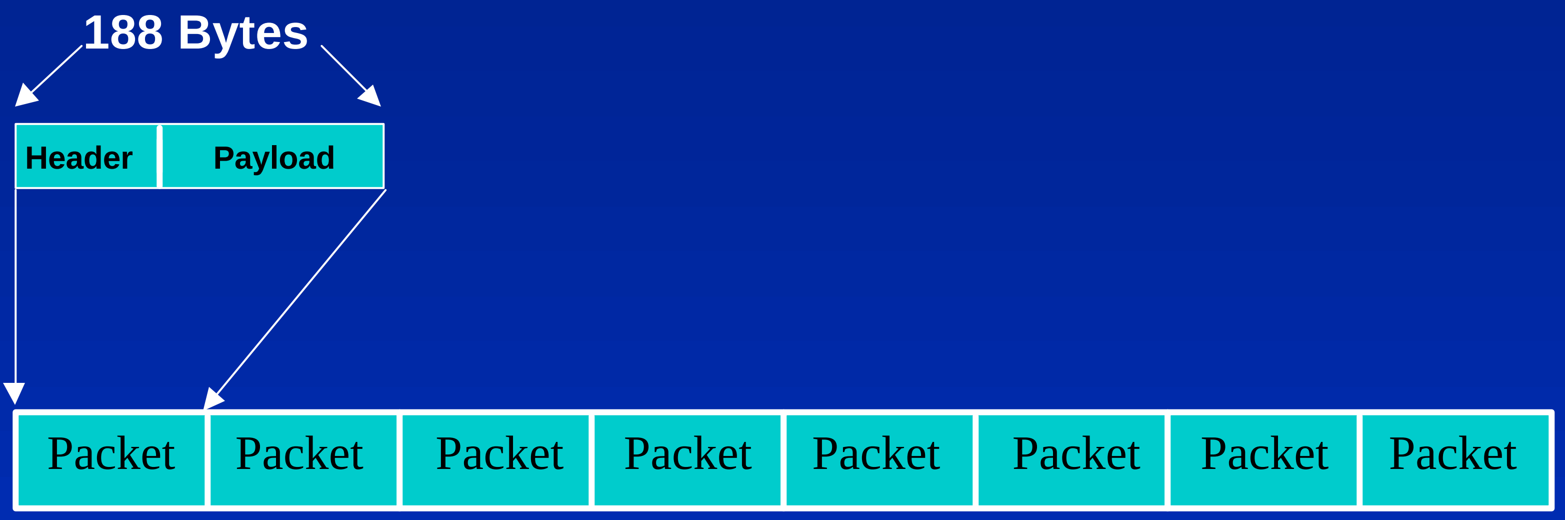
Functional Layers Typical in MPEG



Program Clock Model

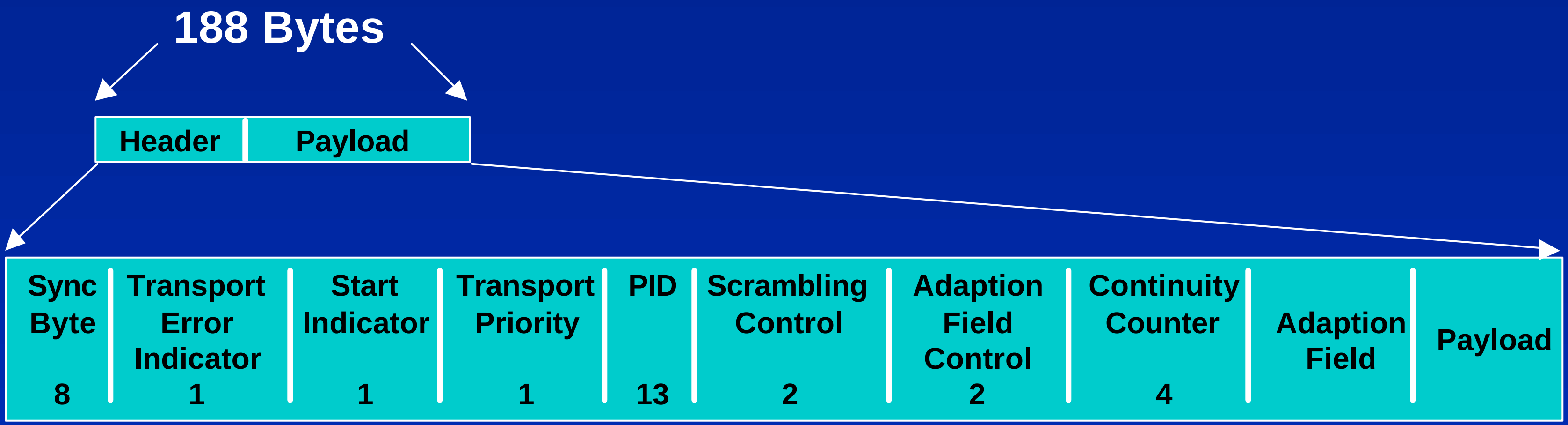

PCR clock frequency generator
PCR clock phase generator
PCR: encoder stamps departure time of packet
PCR: arrival time of packet
PCR clock recovered
Display
video encoder
audio encoder
PES Syntax
system encoder
system decoder
PES Syntax
video encoder
audio encoder
variable delay = e(n)
constant trans delay = C_{trans}
variable delay = d(n)
constant total delay = C_{total}

Transport Layer



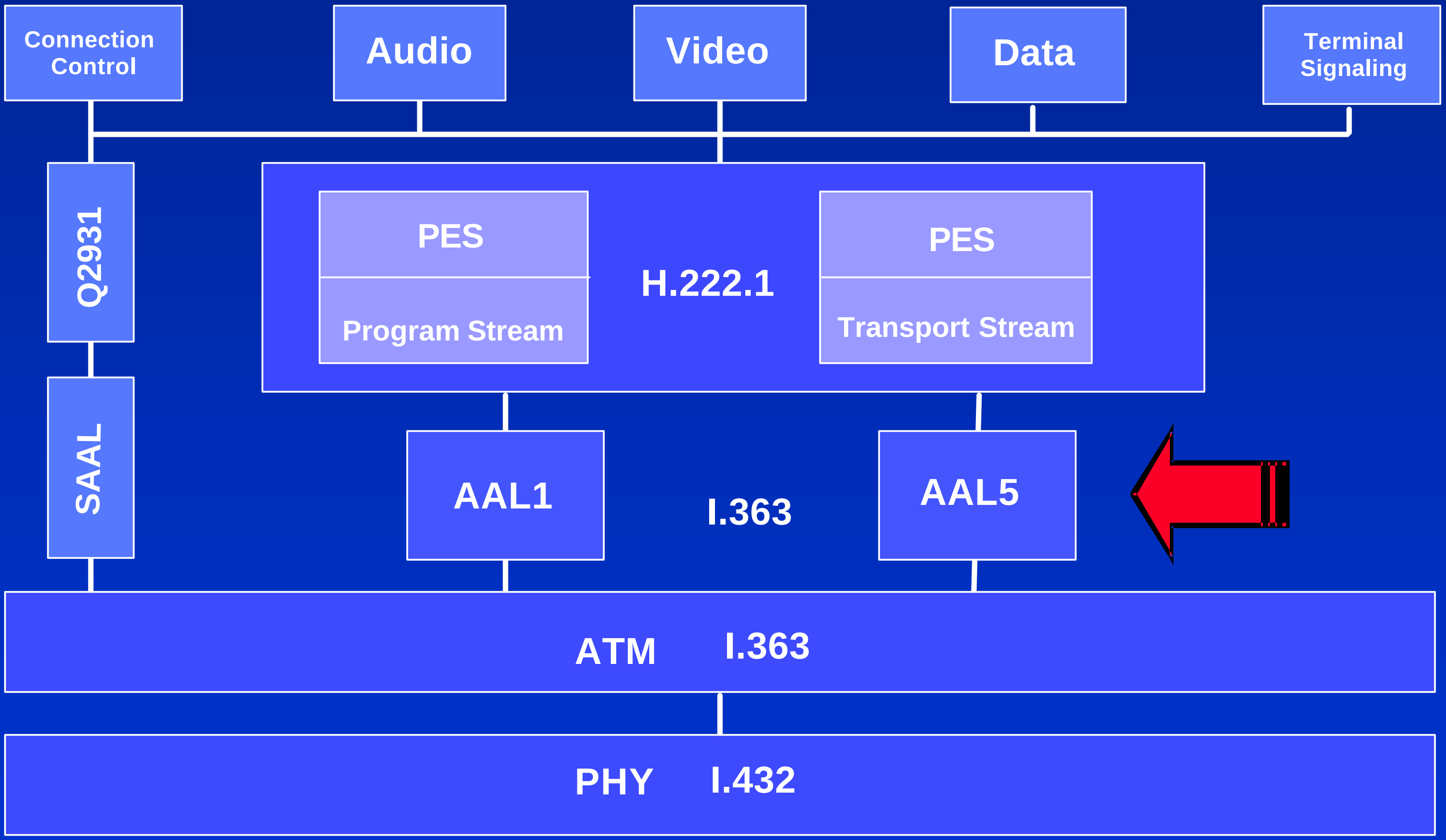
- The Transport Stream (TS) is a continuous data stream in 188 byte packets containing format (syntax) information and payload data

Transport Layer



- Transport Stream PSI
  - Program Association Table (PAT)
  - Program Map Tables (PMT)
  - Conditional Access Table (CAT)
  - Packet Identification (PID)

ATM Multimedia Standards Framework



Hardware Assisted Playback of Compressed Audio.

MPEG Layer 3 Information

You can find a very detailed description (in postscript) at this address: http://www.kom.auc.dk/DSP/Doc/1014_97/ I is the Masters thesis of Kent Salomonsen,Sten Søgaard, and Eddie Proft Larsen. A brief description of the thesis can be found here: http://www.kom.auc.dk/DSP/publics/publics.html. I have a local copy here in postscript, see the links page.

Design and Implementation of an MPEG/Audio Layer III Bitstream Processor
K. Salomonsen, S. Søgaard, and E.P. Larsen, 1997.
Main report, Appendices, Supplement, Companion disk.

Here is my effort at explaining the format. Please note, this page is based on the above explination, the MPEG ISO document, and the reference source code.

Or it will be once I’ve written it. :-)

UNDER CONSTRUCTION

MPEG Audio Frame

Sync Word	Frame Header	Side Info	Main Data	Ancillary Data

Sync Word is: 0xFFF (1111 1111 1111)

Frame Header: 18 bit structure

Side Info: variable bit length structure

Main Data: compressed sound packets.

Ancillary Data: Ignored by decoder, for inserting user defined data into the bitstream, e.g. Song Title etc. (NB. not ID3 tags)

Frame Header

Name	No. Bits	Description
id	1	`=1 for ISO/IC11172-3 audio,` `=0 Reserved`
layer	2	`=11 Layer I,` `=10 Layer II,` **`=01 Layer III`**`,` `=00 Reserved`
protection bit	1	`=0 Error Checking` `=1 No Error Checking.`
bitrate index	4	*See bitrate table*
sampling frequency	2	`=00 44.1 KHz` `=01 48 KHz` `=10 32 KHz` `=11 reserved`
padding bit	1	`=1 extra data added to adjust mean bitrate`
private bit	1	`not used`
mode	2	`=00 stereo` `=01 joint stereo` `=10 dual channel` `=11 single channel`
mode extension	2	*used for joint stereo only - see table*
copyright	1	`=1 Copyrighted`
original / copy	1	`=1 Original`
emphasis	2	`=00 none` `=01 50/15 microseconds` `=10 reserved` `=11 CCITT J.17`

Bitrate Table

Joint Stereo Mode Extension Table

...

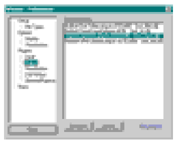
Digital Audio Systems
formerly MPEG Layer 3 Sounds

Information on MP3 decoding.

technical

DECODING

MP3 Players:
Most mp3 players can now be set to output to a file instead of a sound device when playing. In this way they can be used to decode a mp3 to wav. Check the instructions for your player to find out how to do this. In WinAMP you set the output plugin to ’Disk Writer’ and set the output path in [options]. You can decode a group of files by making a playlist. Be sure to set the output plugin back to waveOut or DirectSound when you are done.



Command Line Utility:
If you need a linear PCM file (WAV, AIF, SND or other) from your mp3’s, it’s easy to decode them with Fraunhofer IIS l3dec (which comes with the older versions of l3enc and the registered verison of mp3enc availble on the software page). This program is fully functional in the shareware release, all you need to do is enter the following at the command line and you’ll have a huge file again.

```
l3dec filename.mp3 filename.wav -wav
```

You will recieve a message that says "EOF !" when the program is finished decoding, along with the number of frames and the length - this is not an error, just the message indicating that the program is done. You could also get a lost synchronization error during the last frame depending on the encoder that was originally used - the file should still be fine though.

Digital Audio Systems
formerly MPEG Layer 3 Sounds

Encoding information for MP3 files.

technical

ENCODING

MP3 Encoder software

Sound quality is highly dependent on the performance of the encoder. **Better encoder software will produce higher-quality files with fewer sound problems.** FhG IIS (Fraunhofer-Gesellschaft Institut Integrierte Schaltungen) developed much of the mp3 standard and have some of the best sounding encoders available. There is info on specific encoder programs in the software section.

CBR
Most encoders use constant-bitrate (CBR) encoding. In this mode you choose a target bitrate (say 128kBit/s) and the encoder will hit it*. 4 different stereo modes are available for encoding stereo files:

- **Intensity stereo** - Mono recording with direction information to create a stereo effect.
- **Joint stereo** - Uses intensity mode for low frequency information and discrete channels for high frequency information.
- **Stereo** - Encoders ’share’ unused bandwidth between channels.
- **Dual channel** - Encodes each channel completely separately.

VBR
Xing technologies currently has the only Variable-bitrate coder. VBR files use an index to determine if enough signal has been encoded to meet quality requirements. The index is a number like 50, 75 or 90 or a pnemonic like ’Normal (50)’ or ’High (75)’. The datarate can

be as low as 32kBit/s and as high as necessary - usually averaging 110-140kBit/s in normal mode. The Joint stereo method of encoding is used. As a note, Xing's VBR encoder only works with 44.1kHz source, other samplerates will be converted (the CBR mode of that encoder only works for 44.1/22.05/11.025kHz).

All properly written MP3 players can playback VBR encoded files.

The Fraunhofer codecs are used in the current version of Musicmatch jukebox and a few other programs. Xing MP3 Encoder and AudioCatalyst use VBR. See the software page for more information.

*If a portion of the sound source is not complex enough to fill a given MPEG frame it will be padded out or, in FhG IIS codecs, marked for use in 'high-quality' encoder modes.

Audio sources

To get a high-quality compressed file you need to start with high-quality source material - this usually means CD's (and will soon include uncompressed DVD Audio). To obtain the best files from your CD's it would be advisable to make a direct digital copy using digital audio extraction (called ripping). A surprising number of CD-ROM drives (both IDE and SCSI) can do this. If you would like to see what ripping software will work with your drive (if any) then you need to go to the CDDA page. Check the 'results' page and look for your drive. Tip: if you are running Windows 9x you can get the actual model of your drive from device manager (in the system applet). **If you are looking to purchase a drive just for this application, there is one best choice:** Plextor.

If you can possibly avoid it, do not use analog captures from CD's. The quality really is inferior, even if it doesn't sound like it right away. **A good trick to get a CD-ripper to allow DAE** is to install a new ASPI layer. Some rippers have resources for this on the websites. Otherwise you could find a friend with a copy of Adaptec Easy CD creator and install it, then visit Adaptec and grab the newest update for your OS.

Problems with MP3 encoded files

If you're encoding and run across a file that just doesn't sound good when coded at 112 or 128kbit/s, you can just code at a higher data rate to remove some of the compression artifacts. 160kbit/s is supported by most of the coders, and should eliminate some problems - if not then 256kbit/s will. The 'squishy', 'hollow' or 'flippy' sounds are typically caused by phase shifts in the source file, which are very common in non-professional live stereo recordings, and deliberately present in surround encoded files.

Some compression problems are not due to insufficient datarate - a number of these problems are caused by the use of Joint-Stereo coding, where some frequency bands are encoded in mono (with steering information so it can be reproduced in the correct location in the soundfield). Instead of just increasing the datarate of your files you can try an encoder capable of producing Simple-stereo or 2ch files (dual mono). For these files I recommend using slightly higher datarates: 128, 160 or 192kbit/s to offset the increase in data. The 256kbit/s mode in mp3enc (formerly l3enc) uses 2ch coding and can carry two completely seperate program streams. Most newer coders will use Simple stereo mode at high datarates (including the FhG IIS ones) and should get you that CD-quality sound you’re after.

One more thing: If you are trying to batch encode a sequence of files and the encoder keeps crashing try opening up the file in a sound editor and pad a small amount of silence to the beginning or end and resave it.

Digital Audio Systems
formerly MPEG Layer 3 Sounds

Reference information for mp3, mp4(aac).

technical

REFERENCE

Tiny bits of source code

Encoders
(MP3) A great new project! Here is a link to the new LAME project. This is an open-source coder (no licencing fees!) and is being *actively* developed. The coding engine is alteady in use in some commercial applications.
LAME

(MP3) Here is a .zip file with the original ISO MPEG Audio codec source code. It is somewhat buggy, but gives a good idea of what is needed
Dist10.zip

(AAC) The FACC project is an attempt to get a free coder that is compatible with MPEG-4 TF/AAC Main and Low level.
FAAC at sourceforge

Decoders
(MP3) The only decoder software I can locate currently is included with maPlay. You can obtain it from the homepage. There is some more available from Fraunhofer that I need to locate.
maPlay_v1.A_src.zip

Technical reference material

If you do not have postscript support, you can get Aladdin GhostScript for free. To view pdf's use Adobe Acrobat reader (also free).

document	**formats**
Digital Audio Compression	postscript pdf
A Tutorial on MPEG/Audio Compression	postscript pdf
.zip archive of figures for the above file	postscript pdf
The use of multirate Filter banks for coding of high quality digital audio	postscript

Digital Audio Systems

formerly MPEG Layer 3 Sounds

Setup real-time audio streaming for *your* files.

Save $10 when you grocery shop online
GET $10 OFF HOME DELIVERY NOW!
SAFEWAY

technical

STREAMING

Server Configuration

If you want your files to work properly on your web server you should add (or verify the presence of) the MIME types below. The larger online companies already have them. If they are not there yet, you should not have any trouble convincing your OSP/ISP to include them. If you can create and edit .htaccess files on your account you can add them yourself with the addtype directive:

```
AddType audio/x-mpeg .mp3
AddType audio/x-mpegurl .m3u .mp3url
AddType audio/x-pn-realaudio .ram
```

Streaming Setup

In order to send a `.mp3` file to be played in real-time you will need two things:

- First, the datarate of your compressed file must be compatible with the connection speed of your audience - that means no CD-quality stereo over an analog modem in realtime.
- Second, there must be a way to let the player know where the file is. `.m3u` and `.ram` files were created for this purpose. They are text files that contain the URL of one or more files.

To increase useability you may wish to provide low and high datarate versions of a file, and

offer the ability to download it for offline use.
Now, to make an `.m3u`, create a text file in any (preferably simple) text editor - like the ubiquitous Windows notepad (the universal tool for HTML editing, small Java[tm] programs, batch files, Perl scripts and now m3u's). In this file put a URL such as:

```
http://www.domain.com/~username/mp3files/mymp3file.mp3
```

(note this is not a working URL) and save it as `mymp3file.m3u` ... *that's it*, you now can stream the file over a network. You may want a carriage return (push enter) at the end of the line or it may screwup WinAMP. You can also put more than one URL in the file (then it would be a playlist!).
RealAudio files can be done the same way just use a `.ra` file and name your text file `.ram` - this will setup http-based realaudio streaming. If your webserver has a RealAudio server you may want to use `pnm://` (progressive networks multicast) instead of `http://` in your .ram files - this will instruct RealPlayer to connect to your multicast server and try to save you bandwidth.

Note: though most servers do not map the extension .mp3url to MIME:audio/x-mpegurl, it is the same as .m3u.

The sounds section contains **streaming links for every file**, though you'll need *at least* dual-ISDN or IDSL for most of them. You can also check out some sample low datarate files.

10 RATE CONTROL AND QUANTIZATION CONTROL

This section describes the procedure for controlling the bit-rate of the Test Model by adapting the macroblock quantization parameter ***quantizer_scale***. The algorithm works in three-steps:

1 Target bit allocation: this step estimates the number of bits available to code the next picture. It is performed before coding the picture.

2 Rate control: by means of a "virtual buffer", this step sets the reference value of the quantization parameter for each macroblock.

3 Adaptive quantization: this step modulates the reference value of the quantization parameter according to the spatial activity in the macroblock to derive the value of the quantization parameter, mquant, that is used to quantize the macroblock.

Step 1 - Bit Allocation

Complexity estimation

After a picture of a certain type (I, P, or B) is encoded, the respective "global complexity measure" (Xi, Xp, or Xb) is updated as:

$$X_i = S_i Q_i,$$
$$X_p = S_p Q_p,$$
$$X_b = S_b Q_b,$$

where Si, Sp, Sb are the number of bits generated by encoding this picture and Qi, Qp and Qb are the average quantization parameter computed by averaging the actual quantization values used during the encoding of the all the macroblocks, including the skipped macroblocks.

Initial values

Xi = (160 * **bit_rate**) / 115

Xp = (60 * **bit_rate)** / 115

Xb = (42 * **bit_rate**) / 115

bit_rate is measured in bits/s.

Picture Target Setting

The target number of bits for the next picture in the Group of pictures (Ti, Tp, or Tb) is computed as:

$$T_i = \max\left\{ \frac{R}{\left(1 + \frac{N_p X_p}{X_i X_p} + \frac{N_b X_b}{X_i K_b}\right)}, \frac{bit\_rate}{8 \times picture\_rate} \right\}$$

$$T_p = \max\left\{ \frac{R}{\left(N_p + \frac{N_b K_p X_b}{K_b X_p}\right)}, \frac{bit\_rate}{8 \times picture\_rate} \right\}$$

$$T_b = \max\left\{ \frac{R}{\left(N_b + \frac{N_p K_b X_p}{K_p X_b}\right)}, \frac{bit\_rate}{8 \times picture\_rate} \right\}$$

Where:

Kp and Kb are "universal" constants dependent on the quantization matrices. For the matrices specified in sections 7.1 and 7.2 Kp = 1.0 and Kb = 1.4.

R is the remaining number of bits assigned to the GROUP OF PICTURES. R is updated as follows:

After encoding a picture , R = R - Si,p,b

Where is Si,p,b is the number of bits generated in the picture just encoded (picture type is I, P or B).

Before encoding the first picture in a GROUP OF PICTURES (an I-picture):

R = G + R

G = **bit_rate** * N / **picture_rate**

N is the number of pictures in the GROUP OF PICTURES.

At the start of the sequence R = 0.

Np and Nb are the number of P-pictures and B-pictures remaining in the current GROUP OF PICTURES in the encoding order.

I	B	B	P	B	B	P	B	B	P	B	B	P
						R-bits						
						Np = 3						
						Nb = 4						

Figure 10.1 - Example of Remaining pictures in GOP at frame 7

Step 2 - Rate Control

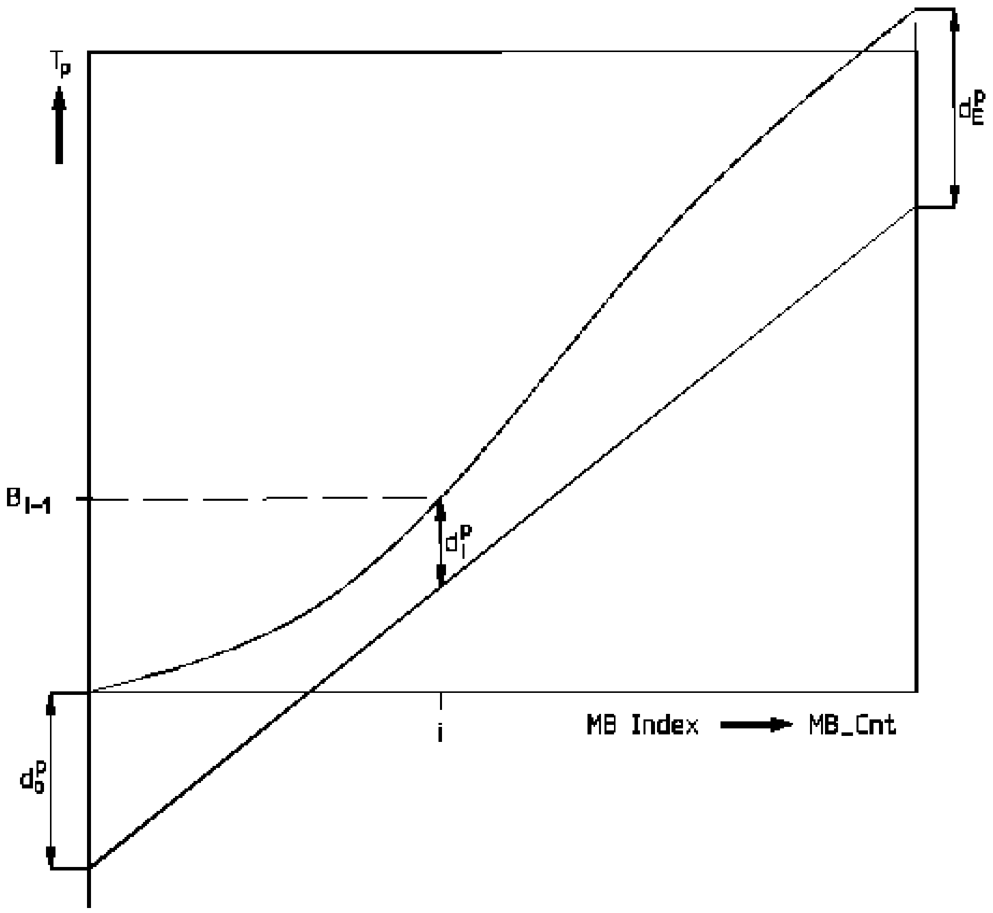


Figure 10.2 : Rate Control for P-pictures

Before encoding macroblock j (j >= 1), compute the fullness of the appropriate virtual buffer:

$$d_j^i = d_0^i + B_{j-1} - \left(\frac{T_i \times (j-1)}{MB\_cnt} \right)$$

or

$$d_j^p = d_0^p + B_{j-1} - \left(\frac{T_p \times (j-1)}{MB\_cnt} \right)$$

or

$$d_j^b = d_0^b + B_{j-1} - \left(\frac{T_b \times (j-1)}{MB\_cnt} \right)$$

depending on the picture type.

where

d_0^i, d_0^p, d_0^p are initial fullnesses of virtual buffers - one for each picture type.

Bj is the number of bits generated by encoding all macroblocks in the picture up to and including j.

MB_cnt is the number of macroblocks in the picture.

d_j^i, d_j^p, d_j^p are the fullnesses of virtual buffers at macroblock j- one for each picture type.

The final fullness of the virtual buffer (d_j^i, d_j^p, d_j^p : j = MB_cnt) is used as d_0^i, d_0^p, d_0^p for encoding the next picture of the same type.

Next compute the reference quantization parameter Qj for macroblock j as follows:

$$Q_j = \left(\frac{d_j \times 31}{r} \right)$$

where the "reaction parameter" r is given by

$$r = 2 \times \frac{bit\_rate}{picture\_rate}$$

and dj is the fullness of the appropriate virtual buffer.

The initial value for the virtual buffer fullness is:

$$d_0^i = 10 \times \frac{r}{31}$$
$$d_0^p = K_p \times d_0^i$$
$$d_0^b = Kb_b \times d_0^i$$

Step 3 - Adaptive Quantization

Compute a spatial activity measure for the macroblock j from the four luminance frame-organised sub-blocks (n=1..4) and the four luminance field-organised sub-blocks (n=5..8) using the intra (i.e. original) pixel values:

$$act_j = 1 + \min(\, vblk_1, vblk_2, \ldots, vblk_8)$$

where

$$vblk_n = \frac{1}{64} \times \sum_{k=1}^{64} \left(P_k^n - P\_mean_n \right)^2$$

and

$$P\_mean_n = \frac{1}{64} \times \sum_{k=1}^{64} P_k^n$$

and Pk are the sample values in the n-th original 8*8 block.

Normalise actj:

$$N\_act_j = \frac{\left(2 \times act_j\right) + avg\_act}{act_j + \left(2 \times avg\_act\right)}$$

avg_act is the average value of actj the last picture to be encoded. On the first picture, avg_act = 400.

Obtain mquantj as:

$$mquant_j = Q_j \times N\_act_j$$

where Qj is the reference quantization parameter obtained in step 2. The final value of mquantj is clipped to the range [1..31] and is used and coded as described in sections 7, 8 and 9 in either the slice or macroblock layer.

Known Limitations

- Step 1 does not handle scene changes efficiently.
- A wrong value of avg_act is used in step 3 after a scene change.
- VBV compliance is not guaranteed.

- Back to Contents

5 MOTION ESTIMATION AND COMPENSATION

To exploit temporal redundancy, motion estimation and compensation are used for prediction.

Prediction is called forward if reference is made to a frame in the past (in display order) and called backward if reference is made to a frame in the future. It is called interpolative if reference is made to both future and past.

For this TM the search range should be appropriate for each sequence, and therefore a vector search range per sequence is listed below:

Sequence	Frame vertical range	Field vertical range	Horizontal range
Table Tennis	± 15 samples	± 3 samples	± 7 samples
Flower Garden	± 15 samples	± 3 samples	± 7 samples
Calendar	± 15 samples	± 3 samples	± 7 samples
Popple	± 15 samples	± 3 samples	± 7 samples
Football	± 31 samples	± 7 samples	± 15 samples
PRL Car	± 63 samples	± 15 samples	± 31 samples

A positive value of the horizontal or vertical component of the motion vector signifies that the prediction is formed from pixels in the referenced frame, which are spatially to the right or below the pixels being predicted.

5.1 Motion Vector Estimation

For the P and B-frames, two types of motion vectors, Frame Motion Vectors and Field Motion Vectors, will be estimated for each macroblock. In the case of Frame Motion Vectors, one motion vector will be generated in each direction per macroblock, which corresponds to a 16x16 pels luminance area. For the case of Field Motion Vectors, two motion vectors per macroblock will be generated for each direction, one for each of the fields. Each vector corresponds to a 16x8 pels luminance area.

The algorithm uses two steps. First a full search algorithm is applied on original pictures with full pel accuracy. Second a half pel refinement is used, using the local decoded picture.

5.1.1 Full Search

A simplified Frame and Field Motion Estimation routine is listed below. In this routine the following relation is used:

```
  (AE of Frame) = (AE of FIELD1) + (AE of FIELD2)

where AE represents a sum of absolute errors.
```

```
With this routine three vectors are calculated, MV_FIELD1, MV_FIELD2 and MV_FRAME.

  Min_FRAME  = MAXINT;

  Min_FIELD1 = MAXINT;

  Min_FIELD2 = MAXINT;

  for (y = -YRange; y < YRange; y++)
  {

    for (x = -XRange; x < XRange; x++)
    {
      AE_FIELD1 = AE_Macroblock( prediction_mb(x,y), lines_of_FIELD1_of_current_mb )
      AE_FIELD2 = AE_Macroblock( prediction_mb(x,y), lines_of_FIELD2_of_current_mb )

      AE_FRAME = AE_FIELD1 + AE_FIELD2;

      if (AE_FIELD1 < Min_FIELD1)
      {
        MV_FIELD1 = (x,y);
        Min_FIELD1 = AE_FIELD1;
      }

      if (AE_FIELD2 < Min_FIELD2)
      {
        MV_FIELD2 = (x,y);
        Min_FIELD2 = AE_FIELD2;
      }

      if (AE_FRAME < Min_FRAME)
      {
        MV_FRAME = (x,y);
        Min_FRAME = AE_FRAME;
      }
    }
  }
```

The search is constrained to take place within the boundaries of the significant pel area. Motion vectors which refer to pixels outside the significant pel area are excluded.

5.1.2 Half pel search

The half pel refinement uses the eight neighbouring half-pel positions in the referenced corresponding local decoded field or frame which are evaluated in the following order:

1	2	3
4	0	5
6	7	8

where 0 represents the previously evaluated integer-pel position. The value of the spatially interpolated pels are calculated as follows:

S(x+0.5,y) = (S(x,y)+S(x+1,y))//2,

S(x ,y+0.5) = (S(x,y)+S(x,y+1))//2,

=

S(x+0.5,y+0.5) = (S(x,y)+S(x+1,y)+S(x,y+1)+S(x+1,y+1))//4.

where x, y are the integer-pel horizontal and vertical coordinates, and S is the pel value. If two or more positions have the same total absolute difference, the first is used for motion estimation.

NOTE: In field searches, the refence system is the correspondig field. In a field the line distance is 1.

5.1.3 Motion estimation for Special prediction mode

The first step is to obtain four candidate motion vectors as follows :

First, four field motion vectors with half-pel accuracy from reference field 1 / field 2 to predicted field 1 / field 2 are searched by normal motion vector search defined in the Test Model. Then these vectors are appropriately scaled, if the parity of the predicted field is opposite to that of the predicted field.

The second step is to evaluate the prediction errors of Dual-prime prediction using possible combinations of four candidate motion vectors obtained by the first step, and 3Vx3H = 9 candidate differential motion vectors.

The prediction error is computed using the reconstructed pictures. The combination with the smallest MSE is selected.

5.2 Motion Compensation

Motion compensation is performed differently for field coding and for frame coding. General formulas for frame and field coding are listed below.

Forward motion compensation is performed as follows:

S(x, y) = S1(x + FMVx(x, y), y + FMVy(x, y))

Backward motion compensation is performed as follows:

S(x, y) = SM+1(x + BMVx(x,y), y + BMVy(x,y))

Temporal interpolation is performed by averaging.

S(x,y) = (S1(x + FMVx(x,y) , y + FMVy(x,y)) + SM+1(x + BMVx(x,y), y + BMVy(x,y)))//2

where FMV is the forward motion compensated macroblock, thus making reference to a ’previous picture’, and BMV is the backward motion compensated macroblock, making reference to a ’future picture’.

A displacement vector for the chrominance is derived by halving the component values of the corresponding MB vector, using the formula from CD 11172, section ......:

```
 right_for                    = (recon_right_for / 2) >> 1;
```

down_for	= (recon_down_for / 2) >> 1;
right_half_for	= recon_right_for/2 - 2*right_for;
down_half_for	= recon_down_for/2 - 2*down_for;

5.2.1 Frame Motion Compensation

In frame prediction macroblocks there is one vector per macroblock. Vectors measure displacements on a frame sampling grid. Therefore an odd-valued vertical displacement causes a prediction from the fields of opposite parity. Vertical half pixel values are interpolated between samples from fields of opposite parity. Chrominance vectors are obtained directly by using the formulae above. The vertical motion compensation is illustrated in figure 5.1.

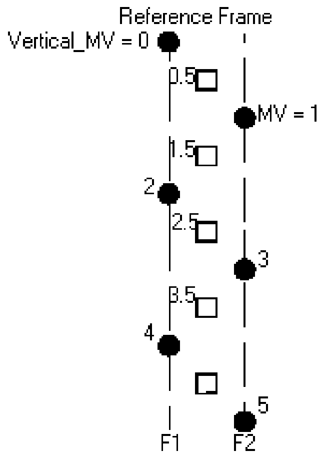


Figure 5.1: Frame Motion Compensation

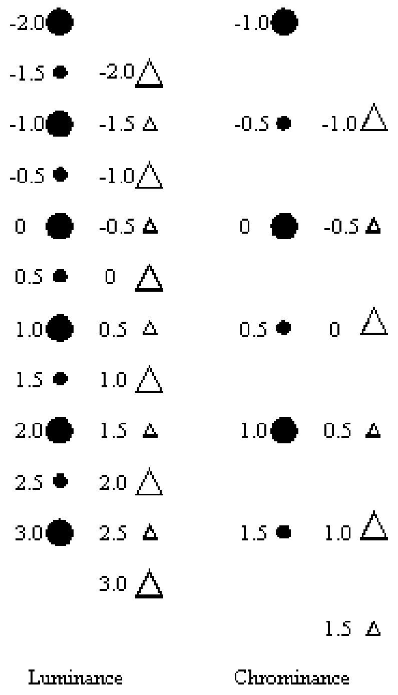
5.2.2 Field Motion Compensation

Field-based MV is expressed in the very same way as frame-based vectors would be if the source (reference) field and the destination field were considered as "frames" (see Figure).

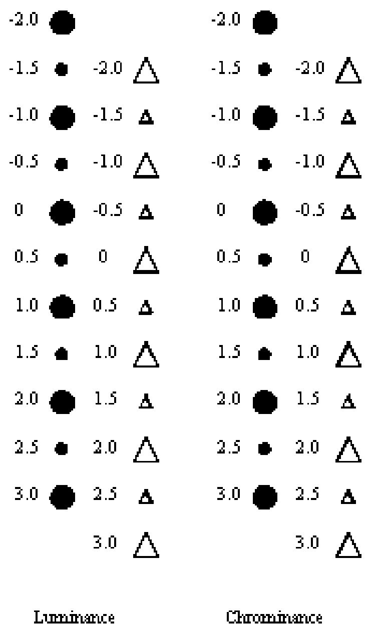
Considering that in each field, lines are numbers 1.0, 2.0, 3.0, ... (1 is the top line of the field), if the pel located at line "n" of the destination field is predicted from line "m" of the reference field, the vertical coordinate of the field vector is "n-m".

Note: when coding the motion vectors, "m" and "n" are expressed in units of one vertical half-pel in the field.

When necessary, **motion_vertical_field_select** (one bit) will be transmitted to identify the selected field.



4:2:0 format



4:2:2 format and 4:4:4 format

5.2.2.1. Chrominance Field-based MV

In 4:2:0 sequences :

The vertical coordinate of the chrominance Field-based MV is derived by dividing by 2 the vertical coordinate of the luminance Field-based MV, as done in MPEG-1.

The horizontal coordinate of the chrominance MV (Field-based or Frame-based) is derived by dividing by 2 the horizontal coordinate of the luminance MV, as done in MPEG-1.

In 4:2:2 sequences :

The vertical coordinate of the Field-based MV for chrominance is equal to the vertical coordinate of the luminanceField-based MV.

The horizontal coordinate of the chrominance MV (Field-based or Frame-based) is derived by dividing by 2 the horizontal coordinate of the luminance MV, as done in MPEG-1.

In 4:4:4 sequences :

The horizontal (resp. vertical) coordinate of MV for chrominance is equal to the horizontal (resp. vertical) coordinate of the luminance MV.

5.3 Special prediction mode

5.3.1. Overview of *Special* Prediction mode

There is only one *special* prediction mode (Dual-prime) remaining in this Test Model and this is based on Field-based prediction. **THIS IS ONLY USED FOR M=1 CODING (NO B_FRAMES) FOR THE MAIN PROFILE, MAIN LEVEL. FOR OTHER PROFILES AND LEVELS IT HAS NOT BEEN DECIDED.** This mode has been included in particular for low delay applications.

Dual Prime prediction involves the averaging of two forward field based predictions from the last two nearest decoded fields (in time).

In the syntax of the Special prediction mode, for forward prediction, one field motion vector is transmitted, followed by a differential motion vector. Each of the coordinates of the differential motion vector is limited to the values [-1, 0, +1] (half pixel values), and is transmitted with a 1-2 bit code.

Combinations of the transmitted field motion vector (possibly scaled according to the field temporal distance) and of the differential motion vector are used for the prediction, as described in the following sections. A separate section defines precisely how field motion vectors are scaled.

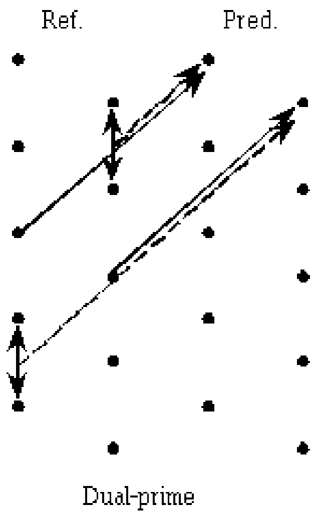


Figure 1 : Special prediction mode (frame structure picture coding mode)

Plain arrows represent the transmitted field motion vector. Dashed arrows represent the scaled-up or scaled-down field motion vectors. Vertical arrows represent the transmitted differential motion vector.

5.3.2. Specification of Dual-prime vectors

Motion vectors used for Dual-prime prediction are field motion vectors obtained as follows:

1. If the reference field and the predicted field are same parity, the field motion vector used is equal to the transmitted field motion vector.

2. If the reference field and the predicted field are different parity, the field motion vector used is obtained by adding the differential motion vector to the scaled transmitted motion vector.

NOTE: that the same differential motion vector is used for the scaled-down and the scaled-up field motion vectors.

5.3.3. Temporal Scaling of the Field Motion Vector

The transmitted field motion vector (x, y) corresponds to the temporal distance between two fields of same parity. The horizontal and vertical coordinate are in 1/2-pel units.

The transmitted field motion vector is used for computing two scaled field motion vectors that serve in the Special prediction mode when reference field and predicted fields are opposite parity. One of the scaled field motion vectors is longer ("scaled-up"), the other one is shorter ("scaled-down").

Scaling is done as follows :

If the same parity reference frame is at a distance of 2*k fields from the predicted field, the coordinates (x’, y’) of the scaled motion vector used for accessing the different-parity field is computed as follows:

x’ = (x * K) // 32 (x and x’ are integers)
y’ = ((y * K) // 32) + e (y and y’ are integers)

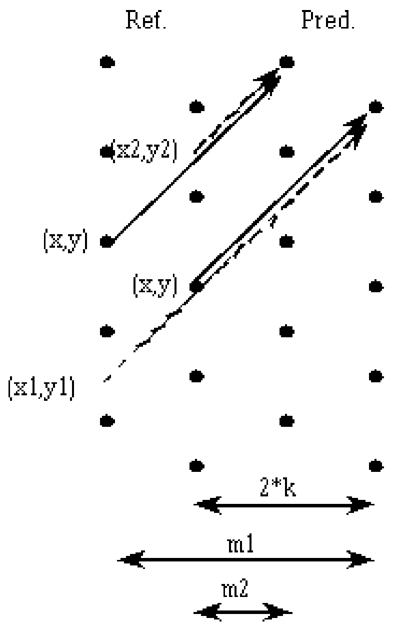
K = (m * 16) // k (k is integer)

m = field-distance between the predicted field and the different-parity-field. NOTE: FURTHER APPROXIMATION OF SCALING SHALL BE REDEFINED(See MPEG93/227)

The "e" is an adjustment necessary to reflect the vertical shift between the lines of field 1 and field 2. To give an example, line 1 of field 2 is in fact located 1/2 line under line 1 of field 1.

e is defined as follows :

e = -1 if the reference field corresponding to the scaled vector is field 2
e = +1 if the reference field corresponding to the scaled vector is field 1



[NOTE: The formula assumes frame based coding and will be updated]

5.3.4. Prediction of Chrominance Blocks

The motion vector used for chrominance is obtained from the luminance Dual-prime motion vector with precisely the same rule as in the case of field-based prediction (for 4:2:0 : divide each coordinate by 2 as described section 5.2.2.1. of TM). The rules of prediction are same as for lumanance.

- Back to Contents

6 MODES AND MODE SELECTION

In section 6.1, a coding structure with different picture modes is introduced. Within each picture, macroblocks may be coded in several ways, thus aiming at high coding efficiency. The MB modes for intra, predicted and interpolated pictures are shown in 6.2 to 6.4.

6.1 Picture types

Pictures are coded in several modes as a trade-off between coding efficiency and random accessibility. There are basically three picture coding modes, or picture types:

- I-pictures: intra coded pictures.

- P-pictures: forward motion-compensated prediction pictures.

- B-pictures: motion compensated interpolation pictures.

Although, in principle, freedom could be allowed for choosing one of these methods for a certain picture, for the Test model a fixed, periodic structure is used depending on the respective picture.

Every N-th picture of a sequence starting with the first picture is coded as intra picture i.e. pictures 1, N+1, etc. (see Test Model Fig. 10.1). Following every M-th picture in between (within a Group of Pictures) is a predicted picture coded relative to the previous predicted or intra picture. The interpolated pictures are coded with reference to the two closest previous and next predicted or intra pictures. In this TM, M=3 and N=15 for 29.97 Hz and M=3 and N=12 for 25 Hz.

The following parameters are currently to be used for most of the core- experiments:

Picture rate	25 Hz	29.97 Hz
N M	12 3	15 3

Coding modes available for predicted and interpolated pictures are described in detail in the following paragraphs.

6.2 Macroblock type decision

Use the MSE critium to select the best Macroblock mode.

6.2.1 Modification of Decision for Field-based Prediction

In order to take advantage of the Special prediction modes, the decision rule must be modified for Field-based prediction.

It has been noted that quality is improved by choosing Field-based prediction less often, to the benefit of the Special prediction mode, particularly in B-Pictures.

For example, even in cases where Field-based prediction has an MSE slightly better than any of the other prediction modes, it may cost a significant overhead to transmit two field-vectors (four in B-Frames).

Until further improvement, we propose to use the following decision rule in core experiments involving the Special prediction modes :

- Field-based prediction chosen

in B-pictures: if (MSE_field + 8 < MSE_best_of_other_modes)

in P-Pictures: if (MSE_field < MSE_best_of_other_modes)

where MSE = Mean Square Error PER PEL of predicted MB

- Back to Contents

7 TRANSFORMATION AND QUANTIZATION

While mode selection and local motion compensation are based on the macroblock structure, the transformation and quantization is based on 8*8 blocks.

Blocks are transformed with a 2-dimensional DCT as explained in Appendix A. Each block of 8*8 pixels thus results in a block of 8*8 transform coefficients. The DCT coefficients are quantized as described in sections 7.1 and 7.2.

7.1 Quantization of Intra Macroblocks

Intra frame DCT coefficients are quantized with selected quantizers without a dead-zone.

7.1.1 DC Coefficients

The quantizer step-size for the DC coefficient of the luminance and chrominance components is 8, 4, 2 and 1. Thus, the quantized DC value, QDC, is calculated as:

```
 QDC          = dc // 8
 QDC(9bit)  = dc // 4
 QDC(10bit)= dc // 2
```

where "dc" is the 11-bit unquantized value from the DCT.

7.1.2 AC Coefficients

AC coefficients ac(i,j) are first quantised by individual quantisation factors,

ac~(i,j) = (16 * ac(i,j)) //wI(i,j)

where wI(i,j) is the (i,j)th element of the Intra quantizer matrix given in figure 7.1. ac~(i,j) is limited to the range [-2048, 2047].

8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	46	56	69
27	29	35	38	46	56	69	83

Figure 7.1 - Intra quantizer matrix

The step-size for quantizing the scaled DCT coefficients, ac~(i,j), is derived from the quantization parameter, quantizer_scale. A quantizer_scale is calculated for each macroblock by the algorithm defined in Test Model Section 10 and is stored in the bitstream in the slice header and, optionally, in any macroblock.

The quantized level QAC(i,j) is given by:

QAC(i,j) = [ac~(i,j) + sign(ac~(i,j))*((p * quantizer_scale) // q)] / (2*quantizer_scale)

If the tcoef_escape_format flag is set to 0, QAC (i,j) is limited to the range [-255..255].

If the tcoef_escape_format flag is set to 1, QAC (i,j) is limited to the range [-2047 .. 2047].

For this TM p=3, and q = 4.

7.2 Quantization Non Intra Macroblocks

Non-intra macroblocks in Predicted and Interpolated pictures are quantized with a uniform quantizer that has a dead-zone about zero. A non-intra quantizer matrix, given in figure 7.2, is used.

16	17	18	19	20	21	22	23
17	18	19	20	21	22	23	24
18	19	20	21	22	23	24	25
19	20	21	22	23	24	26	27
20	21	22	23	25	26	27	28
21	22	23	24	26	27	28	30
22	23	24	26	27	28	30	31
23	24	25	27	28	30	31	33

Figure 7.2 - Non-intra quantizer matrix

The step-size for quantizing both the scaled DC and AC coefficients is derived from the quantization parameter, quantizer_scale. quantizer_scale is calculated for each macroblock by the algorithm defined in Section 10. The following formulae describe the quantization process. Note that INTRA type macroblocks in predicted and interpolated pictures are quantized in exactly the same manner as macroblocks in Intra-pictures (section 7.1) and not as described in this section.

ac~(i,j) = (16 * ac(i,j)) // wN(i,j)

where:

wN(i,j) is the non-intra quantizer matrix given in figure 7.2

QAC(i,j) = ac~(i,j) / (2*quantizer_scale)

If MPEG1 syntax applies, QAC (i,j) is limited to the range [-255..255].

If MPEG2 syntax applies, no clipping is necessary.

7.3 Dequantization

An encoder is free to perform quantization in any way it wants (i.e. "guess", use a mirror of the decoder, or use a fuzzy quantum neural net quadratic maximum a posteri prediction estimator with entropy constrained trellis coded feedback). However, the decoder is deterministic and the rules in IS 13818-2 section 7.4 must be followed.

- Back to Contents

8 CODING

This section describes the coding methods used to code the macroblock header attributes (side information) and data in each macroblock.

For Main Profile, the test model assumes:
- frame structured pictures
- linear macroblock quantizer_scale (q_scale_type==0)
- Zig-Zag scan (alt_scan==0)
- frame_pred_frame_dct==0

The overall syntax and semantics of video decoding is described in the MPEG video standard document (IS 13818-2). The encoder should produce bitstreams knowing the rules of reconstruction in the decoder. Therefore, whenever possible, proceedures for encoding that can be easily derived as the **mirror of the decoding process** are not described here.

In light of the comprehensive and authoratitive manner in which the decoder document describes macroblock reconstruction rules, this section seems redundant. You can almost know how to create a macroblock by following how the decoder unravels it. But this section serves as a guide-by-the-hand walk through the coding process.

The relative horizontal spatial position of each macroblock is encoded by a variable length code, the *macroblock_address_increment* or as we shall call it by its abbreviation: MBA. The use of macroblock addressing is described in section 8.1 of this document and section 6.3.1.17 of IS 13818-2

Macroblocks may take on one of a number of different modes. The modes available depend on the picture type and other high-layer side information (e.g. progressive_sequence) found in the sequence and picture headers. Section 6 describes the procedures used by the encoder to decide on which mode to use. The mode selected is identified in the bitstream by a variable length code known as *macroblock_type*. The use of macroblock_type is described in section 8.2 of the Test Model document, and the general semantics are described in section 6.3.17.1 of IS 13818-2.

The coding of motion vectors is addressed in section 8.3. The decoder counterpart is described in section 7.6.3 of IS 13818-2

Some blocks do not contain any DCT coefficient data. To transmit which blocks of a macroblock are coded and which are non-coded, the coded block pattern (CBP) variable length code is used (see section 8.4).

The coefficients in a block are coded with VLC tables as described in section 8.5, 8.6, and 8.7. The VLC tables are formally given in Annex B of IS 13818-2.

For additional information about frequency and spatially scalable bitstreams, see to Annex D, G and I of the Test Model document. [many of the material is outdated]

8.1 Macroblock Addressing

Relative addressing is used to code the position of all macroblocks in all pictures. Macroblocks for which no data is stored are run-length encoded using the MBA; these macroblocks are called *skipped macroblocks*.

See sections 6.3.1.7 and 7.6.6 of IS 13818-2. Other subsections in IS 13818-2 section 7 describe the semantics of skipped macroblocks, since this mode has a distributed affect throughout the decoder stages (IDCT, motion vectors, DC prediction, etc.).

A macroblock address (MBA) is a variable length code word indicating the position of a macroblock within a MB-Slice. The order of macroblocks is top-left to bottom-right in raster-scan order and is shown in Figure 6-9 of IS 13818-2. For the first non-skipped macroblock in a macroblock slice, MBA is the macroblock count from the left side of the picture. For the Test Model this corresponds to the absolute address in figure 4.3. For subsequent macroblocks, MBA is the difference between the absolute addresses of the macroblock and the last non-skipped macroblock. The code table for MBA is given in Table B.1 of IS 13818-2

The *macroblock_escape* element is a fixed bit-string "0000 0001 000" which is used when the difference macroblock_address_increment is greater then 33. It causes the value of macroblock_address_increment to be 33 greater than the value that will be decoded by subsequenct macroblock_escapes and the macroblock_address_increment codewords.

For example, if there are two macroblock_escape codewords preceding the macroblock_address_ increment, then 66 is added to the value indicated by macroblock_address_increment.

An extra code word is available in the table for bit stuffing immediately after a macroblock slice header or a coded macroblock (MBA Stuffing). This code word should be discarded by decoders.

8.2 Macroblock Type

As described in Table 6-12 of IS 13818-2. each picture has one of the three *picture_coding_type* modes, each of which has a corresponding VLC tables for macroblock_type:

picture_coding_type	picture type	IS 13818-2 table
1	Intra (I-pictures)	B.2
2	Predicted (P-pictures)	B.3
3	Bi-Directional/Interpolated (B-pictures)	B.4

Methods for mode decisions are described in section 6. In macroblocks that modify the quantizer control parameter *quantizer_scale*, the macroblock_type code word is followed by a 5-bit number giving the new value of the quantization parameter denoted by the quantizer_scale_code in the range [1..31].

8.2.1 Compatible Prediction

See Appendix G and J.
[Appendex G and J no longer valid]

8.3 Motion Vectors

Motion vectors for predicted and interpolated pictures are coded differentially within a macroblock slice, obeying the rules established in section 7.6.3 of IS 13818-2. In particular, note that:

- Every forward or backward motion vector is coded relative to the last vector of the same type. Each component of the vector is coded independently, the horizontal component first and then the vertical component.

- The prediction motion vector is set to zero in the macroblocks at the start of a macroblock slice, or if the last macroblock was coded in the intra mode. (**Note**: that in P pictures a No MC, i.e. *macroblock_motion_forward*==0, macroblock_type decision corresponds to a reset to zero of the prediction motion vector.)

- In interpolative pictures, only vectors that are used for the selected prediction mode (MB type) are coded. Only vectors that have been coded are used as prediction motion vectors.

The VLC used to encode the differential motion vector data depends upon the range of the vectors. The maximum range that can be represented is determined by the **forward_f_code** and **backward_f_code** encoded in the picture header. (**Note**: in this Test Model the **full_pel_flag** is never set - all vectors have half-pel accuracy). [half-pel became hardwired in MPEG-2 anyway]

The differential motion vector component is calculated. Its range is compared with the values given in table 8.1 and is reduced to fall in the correct range by the following algorithm:

```
  if (diff_vector < -range)
    diff_vector = diff_vector + 2*range;
  else if (diff_vector> range-1)
    diff_vector = diff_vector - 2*range;
```

forward_f_code or backward_f_code	Range
1	16
2	32
3	64
4	128
5	256
6	512
7	1024

Table 8.1 Range for motion vectors

This value is scaled and coded in two parts by concatenating a VLC found from IS 13818-2 table B.10 and a fixed length part according to the following algorithm:

Let f_code be either the **forward_f_code** or **backward_f_code** as appropriate, and diff_vector be the differential motion vector reduced to the correct range.

```
if (diff_vector == 0)
{
  residual = 0;
  vlc_code_magnitude = 0;
}
else
{
  scale_factor = 1 << (f_code - 1);
  residual = (abs(diff_vector) - 1) % scale_factor;
  vlc_code_magnitude = (abs (diff_vector) - residual) / scale_factor;
  if (scale_factor != 1)
  vlc_code_magnitude += 1;
}
```

The decoder mirror of this equation is given in section 7.6.3.1 of IS 13818-2, albiet with a different notation system. In fact, at the time the Test Model was written (March 1992 - April 1993) before the MPEG-2 video document really gelled in November 1993, so the MPEG-1 style was used.

Test Model/MPEG-1 style	MPEG-2 style	Reason
forward_f_code	f_code[0][t]	concise generalization
backward_f_code	f_code[1][t]	concise generalization
residual	motion_residual[r][s][t]	concise generalization
vlc_code_magnitude	motion_code[r][s][t]	concise generalization
diff_vector	delta	more accurate
scale_factor	f	no reason
PMV1,PMV2,PMV3,PMV4	PMV[r][s][t]	New York meeting in July 1993 cleaned up the notation

See IS 13818-2 table 7.8 for a description of f_code[s][t] and ranges.

vlc_code_magnitude and the sign of diff_vector are encoded according to IS 13818-2 table B.10. The residual is encoded as a fixed length code using (f_code-1) bits.

For example to encode the following string of vector components (measured in half pel units)

3, 10, 30, 30, -14, -16, 27, 24

The range is such that an f value of 2 can be used. The initial prediction is zero, so the differential values are:

3, 7, 20, 0, -44, -2, 43, -3

The differential values are reduced to the range -32 to +31 by adding or subtracting the modulus 64 corresponding to the forward_f_code of 2:

3 7, 20, 0, 20, -2, -21, -3

These values are then scaled and coded in two parts (the table gives the pair of values to be encoded (vlc, residual)):

(2, 0) (4,0) (10, 1) (0, 0) (10, 1) (-1, 1) (-11, 0) (-2, 0)

The order in a slice is in raster scan order, except for Macroblocks coded in Field prediction mode, where the upper two luminance blocks vector are predicted from the preceding Macroblock and the two lower luminance block vectors are predicted according to the rules stated in IS 13818-2 Table 7.9 and Table 7.10.

In MBs that are field DCT coded, chrominance block structure is as follows :

o When the picture format is 4:2:2 and 4:4:4, the chrominance blocks structure is analogous to that of the luminance since the vertical resolution of the picture is the same for luminance and chrominance.

o When the picture format is 4:2:0, the chrominance blocks is structure is equal to that used for frame coded MBs. In other words, chrominance is always frame coded.

Rules for dct_type are stated in IS 13818-2 section 6.1.3

It was agreed that when frame-based prediction is used in non-progressive pictures, the reference field for chrominance prediction may not be the correct one. This slight coding inefficiency is unfortunate, but it was decided for Implementation reasons that the fundamental DCT block size remain 8x8. Other later standards such as Digital Video Cassette have an optional 8x4 DCT block shape.

There are four prediction motion vectors : PMV1, PMV2, PMV3 and PMV4. They are reset to zero at the start of a slice and at intra-coded MBs.

The prediction MVs (PMV1 to PMV4) *are always expressed in Frame MV coordinates* since the Test Model only uses frame structured pictures. See IS 13818-2 section 7.6.3.1 for the influence of *mv_format* on motion vector scaling.

For the prediction of Field-based MVs (mv_format == "field"), the following rules are used:

On the decoder side :

When a Field-based MV is derived, the vertical coordinate of the PMV is shifted right by 1 bit (with sign extension) before adding the vertical differential.

Then the Field-based MV is stored in the appropriate PMV(s) after shifting left by 1 bit its vertical coordinate.

On the encoder side :

When a Field-based MV is encoded, the vertical coordinate of the PMV is shifted right by 1 bit (with sign extension) before it is subtracted from the field MV vertical coordinate.

Then the Field-based MV is stored in the appropriate PMV(s) after shifting left by 1 bit its vertical coordinate.

1. mv_format == "frame" :

In P-Pictures or P-Fields, PMV1 is used. PMV2, PMV3 and PMV4 are reset to PMV1

In B-Pictures or B-Fields, PMV1 is used for forward motion vector prediction, and PMV3 is used for backward motion vector prediction. PMV2 is reset to PMV1, and PMV4 is reset to PMV3.

2. mv_format == "field" :

In P-Frame-Pictures or P-Field-Pictures:

PMV1 is used for vectors used to predict FIELD1 from FIELD1

PMV2 is used for vectors used to predict FIELD1 from FIELD2

PMV3 is used for vectors used to predict FIELD2 from FIELD1

PMV4 is used for vectors used to predict FIELD2 from FIELD2

In B-Picture, PMV1 and PMV2 are used for forward motion vector prediction from field 1 and 2, and PMV3 and PMV4 are used for backward motion vector prediction from fields 1 and 2.

8.3.1 Prediction of Motion Vectors for special prediction modes

When experiments are done involving the Special prediction modes:

In P-Pictures:

PMV1 is used for prediction of the Dual-prime motion vector.
PMV4 is updated with the transmitted field motion vector .
PMV2 is updated with the scaled motion vector from reference field 2 to predicted field 1.
PMV3 is updated with the scaled motion vector from reference field 1 to predicted field 2.

See section 7.6.3.6 for Dual Prime motion vector arithmetic rules

8.4 Coded Block Pattern

See IS 13818-2 section 6.3.17.4

8.5 Intra picture Coefficient Coding

8.5.1 DC Prediction

Decoder rules for DC prediction is described in section 7.2.1 of IS 13818-2.

After the DC coefficient of a block has been quantized to 8 bits according to Test Model section 7.1.1, it is coded loss less by a DPCM technique. Coding of the luminance blocks within a macroblock follows the normal scan of figure 4.4. Thus the DC value of block 4 becomes the DC predictor for block 1 of the following macroblock. Three independent predictors are used, one each for Y, Cr and Cb.

At the left edge of a macroblock slice, the DC predictor is set to 128, 256, 512 and 1024 according to the *intra_dc_precision* variable (for the first block (luminance) and the chrominance blocks). At the rest of a macroblock slice, the DC predictor is simply the previously coded DC value of the same type (Y, Cr, or Cb).

At the decoder the original quantized DC values are exactly recovered by following the inverse procedure.

The differential DC values thus generated are categorised according to their "size" as shown in the table below.

DIFFERENTIAL DC	SIZE
(absolute value)	
0	0
1	1
2 to 3	2
4 to 7	3
8 to 15	4
16 to 31	5
32 to 63	6
64 to 127	7
128 to 255	8
256 to 511	9
512 to 1023	10
1024 to 2047	11

Table 8.2 Differential DC size and VLC

The size value is VLC coded according to IS 13818-2 table B.12 (luminance) and IS 13818-2 table B.13 (chrominance).

For each category enough additional bits are appended to the SIZE code to uniquely identify which difference in that category actually occurred (table 8.3). The additional bits thus define the signed amplitude of the difference data. The number of additional bits (sign included) is equal to the SIZE value.

DIFFERENTIAL DC	SIZE	ADDITIONAL CODE
-2047 to -1024	11	00000000000 to 01111111111
-1023 to -512	10	0000000000 to 0111111111
-511 to -256	9	000000000 to 011111111
-255 to -128	8	00000000 to 01111111
-127 to -64	7	0000000 to 0111111
-63 to -32	6	000000 to 011111
-31 to -16	5	00000 to 01111
-15 to -8	4	0000 to 0111
-7 to -4	3	000 to 011
3 to -2	2	00 to 01
-1	1	0
0	0	
1	1	1
2 to 3	2	10 to 11
4 to 7	3	100 to 111
8 to 15	4	1000 to 1111
16 to 31	5	10000 to 11111
32 to 63	6	100000 to 111111
64 to 127	7	1000000 to 1111111
128 to 255	8	10000000 to 11111111
256 to 511	9	100000000 to 111111111
512 to 1023	10	1000000000 to 1111111111
1024 to 2047	11	10000000000 to 11111111111

Table 8.3. Differential DC additional codes

8.5.2 AC Coefficients

AC coefficients are coded as described in Test Model section 8.7.

8.6 Non-Intrapicture Coefficient Coding

8.6.1 Intra blocks

Intra blocks in non-intra pictures are coded as in intra pictures. At the start of the macroblock, the DC predictors for luminance and chrominance are reset to 128, 256, 512 and 1024 according to the intra_dc_precision, unless the previous block was also intra; in this case, the predictors are obtained from the previous block as in intra pictures (section 8.5.1).

AC coefficients are coded as described in section 8.7. Transform coefficient data is always present for all 6 blocks in a macroblock when macroblock_type indicates *macroblock_intra*==1.

8.6.2 Non intra blocks

In other cases macroblock_type and coded_block_pattern signal which blocks have coefficient data transmitted for them. The quantized transform coefficients are sequentially transmitted according to the zig-zag sequence given in IS 13818-2 Figure 7.2. The Test Model does not use the alternate_scan pattern of IS 13818-2 Figure 7.3

8.7 Coding of Transform Coefficients

First of all there are two VLC’s, one for non intra macroblocks (IS 13818-2 Table B.14), and one for intra macroblocks (IS 13818-2 Table B.15). If the Test Model is applied to create MPEG-1 sequences, only the non intra VLC table is used since MPEG-2’s Table B.14 is the same as MPEG-1’s AC table. The two VLC differ in particular in the length of *end of block* (EOB) code (2 vs. 4 bits, respectively). The combinations of zero-run and the following value are encoded with variable length codes according to IS 13818-2 section 7.2.2. The last bit ’s’ denotes the sign of the level, ’0’ for positive ’1’ for negative.

Blocks with no coefficient data are indicated by the coded_block_pattern, and no EOB is required. Therefore EOB cannot occur as the first coefficient, and hence EOB does not appear in the VLC table for the first coefficient (see note 3 and 4 in IS 13818-2 table B.14).

This first coefficient duality is easiest to comprehend when modelling Table B.14 as really two different tables where only their 2nd entries differs. The first table is used at the very beginning of the block, and then the decoder immediately switches to the second. This kludge saves a considerable number of bits in efficiency.

The approximately 111 most commonly occurring combinations of successive zeros (RUN) and the following value (LEVEL) are encoded with variable length codes listed in the tables. Less common combinations of (RUN, LEVEL) are encoded with a 24-bit escape sequence consisting of a 6 bit ESCAPE code, a 6 bit RUN and a 12 bit LEVEL.

In MPEG-1, the ESCAPE code is followed by a 6 bit run and either an 8 bit or 16 bit level (double escape) depending on the dynamic range of the coefficient.

- Back to Contents

TM5 Overview

The Test Model served as a cook book for creating bitstreams during the collobrative co-exerpimental phase of MPEG-2 video. The many documented experiments were an attempt to verify the usefulness of various proposed coding techniques. Each participant would follow the recipe given in the TM document to create an encoder. If the proposal met the criteria (coding gain, implementation complexity, robustness)--it survived. In the end, only a handful made their way into the MPEG syntax.

The last major update of the Test Model document, version 5, took place at the Sydney, Australia meeting of the MPEG working group (WG11) in March 1993. Since the MPEG-2 main profile syntax froze at this meeting, only limited scalability experiments continued past March 1993. A small two page delta document describing Temporal Scalability experiments, affectionately called "TM 6", was produced at the New York City meeting in July 1993. From then on, direct bitstream exchanges among participants of the working group helped to resolve ambiguities in the official MPEG working draft document which ultimately became the official standard document (ISO/IEC 13818-2, ITU-T H.262) we all know and love today.

History:

The Test Model evolved in parallel with the MPEG video working draft. The TM series was a joint effort between ITU-T SG15.1 (known then as CCITT SG XV, Working Party XV/1, Experts Group on ATM Video Coding) and ISO/IEC JTC1/SC29 WG11 (MPEG).

Version	Meeting location	Date	MPEG doc. no.	ITU-T SG15 doc no.
TM 0	Signapore	Jan 92	[known as Video Working Draft 0 (to keep Cesar	
TM 1	Haifa, Israel	Mar 92	MPEG 92/160	AVC-260
TM 2	Rio, Brazil	Jul 92	MPEG 92/245	AVC-323
TM 2.2	Tarrytown, NY	Oct 92	MPEG 92/535	AVC-356
TM 3	London	Dec 92	WG11 92/328	AVC-400
TM 4	Rome	Feb 93	MPEG 93/225b	AVC-445b
TM 5	Sydney	Apr 93	MPEG 93/457	AVC-491
TM 5b	Sydney	Apr 93	WG11 93/400	AVC-491b
"TM 6"	New York City	Jul 93	-	-

See Leonardo’s nostalgic meeting record for a complete list of meetings.

Parent models:

The first drafts of the coding models inhereted much of the coding methods and documentation style from previous standardization efforts.

```
MPEG-1: final document was SM-3 (Simulation Model version 3)
H.261:  final document was RM-8 (Reference Model version 8)
```

TM 5 Main Profile relevant sections

Sections 1-8, 10, Appendix B, C, K and L of Test Model 5.0 apply to the Main Profile syntax of today.

Of course, corresponding sections in the official MPEG document (IS 13818-2 / ITU-T H.262) always supersede those in the Test Model. Section 9, Appendix D through L are mostly scalability sections, many of which no longer even apply to the scalability syntax of today.

The following is a list of comments on those sections which DO apply to Main Profile:

1 Introduction

The document is no longer maintained. Please do not contact the editor (Koster) unless for historical purposes :-)

2.3.2 Hierarchical profile

- features in "Hierarchical profile" merged into High Profile. - "Harmonized scalable solution" was never found. SNR and Spatial Scalable streams remain separate today.

2.3.3 Professional profile

- features in "professional profile" merged into High Profile.

3.3.1 conversion of CCIR 601 to the 4:2:0 format

Although chroma subsampling is a display process which falls outside the official scope (normative) of the MPEG standard, it should be done properly for quality reasons. The Encoder’s chroma decimator should anticipate the behaviour of the Decoder’s chroma interpolator. The MPEG bitstream header element chroma_420_type was meant to convey which of two methods (interlaced or progressive) a decoder should use to upsample (interpolate) the decoded chroma output. By November 1994, chroma_420_type became synonymous (and therefore redundant) with the progressive_frame header element. chroma_420_type was kept in the syntax for compatibility reasons. After all, the syntax was frozen over 1 year earlier!

Test Model 5 gives two sinc conversion filters suitable for intra-field vertical chroma subsampling in interlaced video:

```
    {-29,0,88,138,88,0,-29}/256  for field 1
    {1,7,7,1}/16                 for field 2

      4:2:2   4:2:0

        Odd

        Even    X

        Odd

        Even    X

        Odd

        Even    X
      Odd

  An inter-field (intra-frame) filter would be used for progressive
  video sequences which renders the subsampled chroma components centered
  between the spatial positions of the original components.

    4:2:2   4:2:0

      Odd
              X
      Even

      Odd
              X
      Even

      Odd
              X
      Even
```

The progressive vs. interlaced vertical chroma filter was later resolved at the November 1993 meeting in Seoul, Korea. Look at section 6.1.1.8 of the final MPEG-2 video document (IS 13818-2 / ITU-T H.262) for a description of interlaced and progressive chroma sample siting.

4. Layered Structure of video data

The official MPEG video document (IS 13818-2 section 6) does an excellent job of describing the layred structure of video data.

5. Motion Estimation and Compensation

See the modified Chapter 5.

6. Modes and Selection

See the modified Chapter 6.

7. Transformation and Quantization

See the modified Chapter 7.

8. Coding

See the modified Chapter 8.

- An additional informative example of encoding is described in Appendix D of the MPEG-1 video document (IS 11172-2).

The following comments point out irrelevant sections with respect to today’s Scalable MPEG-2 syntax:

9. Video multiplex coder

Section 9 never discusses general or Main Profile syntax.

- frequency scalability was dropped from the MPEG toolkit.

- the syntax and semantics for scalability in the MPEG-2 video document supersede those present in this section of TM 5. The are quite different today than they were in April 1993.

10. Rate Control

See the modified Chapter 10.

Appendix A: discrete cosine transform

See the modified Appendix A.

Appendix B: variable length code tables

This section is extremely redundant with Appendix B of IS 13818-2. All the official VLC tables are listed there.

Appendix C: video buffer verifier

The TM only referred to the video specification Working Draft... and rightly so!

Appendix D: frequency domain scalability extension

- again, frequency scalability was dropped from the MPEG toolkit.

Appendix F: cell loss experiments

- this experiment could be generic enough and mostly independent of scalability to apply to Main Profile, although it does refer to jettisoned items such as leaky prediction.

F.4 AC-Leaky Prediction

- leaky prediction was dropped from the MPEG toolkit.

F.5 Data partitioning vs. 1-layer cell loss experiment

- data partitioning is still a part of the MPEG syntax (toolkit), but is yet not enabled in any MPEG Profile.

Appendix G: Compatibility and Spatial Scalability

- the weights for spatial scalablity were modified several times before the final November 1994 video standard draft.

- the only items from Appendix G which would still apply to the scalability syntax are those high-level parameters which specify bitrate, M and N factors, picture types, etc.

Appendix H: Low Delay Coding

- low delay coding is still very much a part of the Main Profile syntax, though is not commonly used. See section D.5 in the MPEG-2 video document for a description.

H.1.6 Influence of leaky prediction on low delay coding

- again, leaky prediction is not a part of the MPEG toolkit.

Appendix I: Frequency Domain Scalability Core Experiments

- again, frequency scalability was dropped from the MPEG toolkit.

Appendix J: Harmonized Hybrid Scalability

- again, no "Unified Field Coding" for scalability was reached. Maybe MPEG-4 will do this. (supressing urge to comment)

Appendix K: Fast Forward and Fast Reverse Modes

- intra_slice is still a legitimate trick mode technique (and about the only one viable today).

K.4 data partitioning approach

- again, data partition is still a part of the MPEG toolkit (cf section 7.10 in IS 13818-2 / ITU-T H.262), but yet not enabled by any Profile and/or Level combination. This section does however demonstrate one of Data Partitioning's primary intended uses.

Appendix L: Data partitioning

- see comments above

- syntax for data partitioning has changed. In particular, there is no data_partitioning_flag, intra_pbp in the sequence and picture headers, respectively. Instead, Data partitioning mode would be signaled by the scalable_mode element (if a tag were defined for it). The pbp element in Appendix L of the TM 5 document is now called priority_breakpoint in the MPEG-2 video document.

Appendix Q: Quantization

- The 8x1 DCT mode was an attempt to solve the ringing artifacts present around "text" and other sharp edges within reconstructed pictures. It is not a part of the MPEG toolkit. Careful coding using existing syntax tools (macroblock *quantizer_scale*) is recommended.

- Back to Contents

Fraunhofer Institut Integrierte Schaltungen

Audio & Multimedia MPEG Audio Layer-3

▶ Fraunhofer AEMT

History

In 1987, the Fraunhofer IIS started to
audio coding in the framework of the
EU147, Digital Audio Broadcasting (
cooperation with the University of Er
Seitzer), the Fraunhofer IIS finally de
algorithm that is standardized as ISO
Layer-3 (IS 11172-3 and IS 13818-3)

Without data reduction, digital aud
consist of 16 bit samples recorded at
than twice the actual audio bandwidth
Compact Discs). So you end up with
Mbit to represent just **one second of**
quality. By using MPEG audio codin
down the original sound data from a
without losing sound quality. Factors
still maintain a sound quality that is s
than what you get by just reducing th
the resolution of your samples. Basic
by *perceptual coding* techniques add
of sound waves by the human ear.

Using MPEG audio, one may achieve
reduction of

1:4	by **Layer 1** (correspond stereo signal),
1:6...1:8	by **Layer 2** (correspond for a stereo signal),
1:10...1:12	by **Layer 3** (correspond for a stereo signal),

still maintaining the original CD sour

By exploiting stereo effects and by li
bandwidth, the coding schemes may
sound quality at even lower bitrates.
most powerful member of the MPEG
For a given sound quality level, it req
bitrate - or for a given bitrate, it achie

quality.

Sound Quality

Some typical performance data of **M**

sound quality	**bandwidth**	**mode**	**b**
telephone sound	2.5 kHz	mono	8
better than short wave	4.5 kHz	mono	1
better than AM radio	7.5 kHz	mono	3
similar to FM radio	11 kHz	stereo	56..
near-CD	15 kHz	stereo	9
CD	>15 kHz	stereo	112.
*) Fraunhofer IIS uses a non-ISO ex Layer-3 for enhanced performance (			

In all international listening tests, MF
impressively proved its superior perfo
the original sound quality at a data re
(around 64 kbit/s per audio channel).
tolerate a limited bandwidth of aroun
reasonable sound quality for stereo si
even at a reduction of 1:24.

For the use of low bit-rate audio codi
broadcast applications at bitrates of 6
channel, the ITU-R recommends MP
doc. BS.1115)

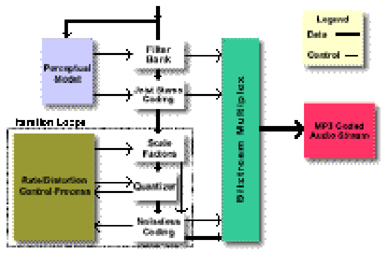


Fig.1: MP3 encoder flowchart

Details

Filter bank

The filter bank used in MPEG Layer-
bank which consists of a polyphase fi
Modified Discrete Cosine Transform
form was chosen for reasons of comp
predecessors, Layer-1 and Layer-2.

Perceptual Model

The perceptual model mainly determ
given encoder implementation. It use
filter bank or combines the calculatio
(for the masking calculations) and the
The output of the perceptual model c
the masking threshold or the allowed
partition. If the quantization noise ca
masking threshold, then the compress
indistinguishable from the original si

Joint Stereo

Joint stereo coding takes advantage o
channels of a stereo channel pair con
information. These stereophonic irrel
redundancies are exploited to reduce
stereo is used in cases where only lov
available but stereo signals are desire

Quantization and Coding

A system of two nested iteration loop
solution for quantization and coding i

Quantization is done via a power-law
way, larger values are automatically c
accuracy and some noise shaping is a
quantization process.

The quantized values are coded by H
specific method for entropy coding, H
lossless. This is called noiseless codi
added to the audio signal.

The process to find the optimum gain
a given block, bit-rate and output fro

model is usually done by two nested
analysis-by-synthesis way:

- **Inner iteration loop (rate loo**

  The Huffman code tables assig
  to (more frequent) smaller qua
  number of bits resulting from t
  exceeds the number of bits ava
  block of data, this can be corre
  global gain to result in a larger
  leading to smaller quantized va
  repeated with different quantiz
  the resulting bit demand for H
  enough. The loop is called rate
  modifies the overall coder rate
  enough.
- **Outer iteration loop (noise co**
  loop)

  To shape the quantization nois
  masking threshold, scalefactor
  scalefactor band. The systems
  factor of 1.0 for each band. If t
  in a given band is found to exc
  threshold (allowed noise) as su
  perceptual model, the scalefact
  adjusted to reduce the quantiza
  achieving a smaller quantizatio
  larger number of quantization
  bitrate, the rate adjustment loo
  every time new scalefactors ar
  the rate loop is nested within t
  The outer (noise control) loop
  actual noise (computed from th
  original spectral values minus
  values) is below the masking th
  scalefactor band (i.e. critical ba

MPEG Software Simulation Group

Wanna know about Dolby and Philips audio war for DVD ?

Visit our *Cool Site of the Month*: MPiG

Realtime MPEG Software Player for Solaris, IRIX & Linux

Download

Support MPEG.ORG by visiting our sponsors

Free MPEG Softwares!

The free MPEG softwares developed by the MPEG Software Simulation Group include an MPEG-2 Video Codec (mpeg2encode and mpeg2decode), a fast MPEG-2 Player (mpeg2play), a fast MPEG-1 Player for Windows (VMPEG-1.7) and an MPEG-1 Player for DOS (VMPEG-1.2).

For pointers to other MPEG Video players and help about installing an MPEG Video Player and configuring your Web browser for playing MPEG files, check our MPEG Video Player page.

And don’t forget to visit the rest of the MPEG.ORG Website!

MPEG-2 Video Codec (ANSI C Source Code)

- Test Model 5 (aka TM5) at MPEG.ORG ***(new)***

A document originating from the MPEG committee. TM5 served as a cook book for creating video bitstreams during the collaborative co-expimental phase of the development of MPEG-2 video. It contains lots of very useful technical informations about MPEG-2 Video encoding techniques. The MSSG MPEG-2 Video encoder software follows the TM5 recipes with only minor differences.

- readme.txt - Archive README file
- mpeg2vidcodec_v12.tar.gz (259,510 Bytes) - Archive Source Code (content)
- mpeg2v12.zip - Archive Source Code + Win32s executables and source (459,421 Bytes) (content)

Older Versions of MPEG-2 Video Codec:

- mpeg2codec_v1.1a.tar.gz (version 1.1a, July 4, 1994: 134,895 Bytes)
- mpeg2codec_verify_v1.1.tar.gz (version 1.1, June 30, 1994: 126,734 Bytes)
- mpeg2play_v1.1b.tar.gz (version 1.1b, July 13, 1994: 36,842 Bytes)

MPEG-2 Video Player for Windows (mpg2w11b)

- mpg2w11b.zip

MPEG-1 Player for Windows (VMPEG-1.7)

VMPEG-1.7 comes as a 550K self-extracting archive for Windows 3.1/95/NT.

- VMPEG-1.7 Announcement - Features, requirements etc.
- VMPEG-1.7 Archive contents

Or use alternate download sites:

- Download VMPEG-1.7 from University of Delaware
- Download VMPEG-1.7 from DVLive
- Download VMPEG-1.7 from ZDNet Software Library
- Download VMPEG-1.7 from Creative.Net

MPEG-1 Programs for DOS

Older VMPEG: fast DOS and Windows 32-bit player (version: 1.2a Date: 94/09/28)
vmpeg12a.zip (154,608 Bytes)

DMPEG: MPEG-1 MS-DOS 32-bit video decode (version: 1.1 Date: 93/06/28)
dmpeg11.zip (28,515 Bytes)

CMPEG: MPEG-1 MS-DOS 32-bit video encoder (version: 1.0 Date: 93/06/28)
cmpeg10.zip (29,692 Bytes)

Utilities

- ieee1180.tar.gz IEEE 1180 mismatch test code for IDCT module (18,782 Bytes)

Contact

You can contact the MPEG Software Simulation Group by email at mssg@mpeg.org.

This is a brief and informal document targeted to those who want to deal with the MPEG format. If you are one of them, you probably already know what is MPEG audio. If not, jump to http://www.mp3.com/ or http://www.layer3.org/ where you will find more details and also more links. This document does not cover compression and decompression algorithm.

NOTE: You cannot just search the Internet and find the MPEG audio specs. It is copyrighted and you will have to pay quite a bit to get the Paper. That's why I made this. Information I got is gathered from the Internet, and mostly originate from program sources I found available for free. Despite my intention to always specify the information sources, I am not able to do it this time. Sorry, I did not maintain the list. :-(

These are not a decoding specs, it just informs you how to read the MPEG headers and the MPEG TAG. MPEG Version 1, 2 and 2.5 and Layer I, II and III are supported, the MP3 TAG (ID3v1 and ID3v1.1) also.. Those of you who use Delphi may find MPGTools Delphi unit (freeware source) useful, it is where I implemented this stuff.

I do not claim information presented in this document is accurate. At first I just gathered it from different sources. It was not an easy task but I needed it. Later, I received lots of comments as feedback when I published this document. I think this last release is highly accurate due to comments and corrections I received.

This document is last updated on December 22, 1999.

MPEG Audio Compression Basics

This is one of many methods to compress audio in digital form trying to consume as little space as possible but keep audio quality as good as possible. MPEG compression showed up as one of the best achievements in this area.

This is a lossy compression, which means, you will certainly loose some audio information when you use this compression methods. But, this lost can hardly be noticed because the compression method tries to control it. By using several quite complicate and demanding mathematical algorithms it will only loose those parts of sound that are hard to be heard even in the original form. This leaves more space for information that is important. This way you can compress audio up to 12 times (you may choose compression ratio) which is really significant. Due to its quality MPEG audio became very popular.

MPEG standards MPEG-1, MPEG-2 and MPEG-4 are known but this document covers first two of them. There is an unofficial MPEG-2.5 which is rarely used. It is also covered.

MPEG-1 audio (described in ISO/IEC 11172-3) describes three Layers of audio coding with the following properties:

- one or two audio channels
- sample rate 32kHz, 44.1kHz or 48kHz.
- bit rates from 32kbps up to 448kbps

Each layer has its merits.

MPEG-2 audio (described in ISO/IEC 13818-3) has two extensions to MPEG-1, usually referred as

MPEG-2/LSF and MPEG-2/Multichannel.

MPEG-2/LSF has the following properties:
- one or two audio channels
- sample rates half those of MPEG-1
- bit rates from 8 kbps up to 256kbps.

MPEG-2/Multichannel has the following properties:
- up to 5 full range audio channels and an LFE-channel (Low Frequency Enhancement <> subwoofer!)
- sample rates the same as those of MPEG-1
- highest possible bitrate goes up to about 1Mbps for 5.1

MPEG Audio Frame Header

An MPEG audio file is built up from smaller parts called frames. Generally, frames are independent items. Each frame has its own header and audio informations. There is no file header. Therefore, you can cut any part of MPEG file and play it correctly (this should be done on frame boundaries but most applications will handle incorrect headers). For Layer III, this is not 100% correct. Due to internal data organization in MPEG version 1 Layer III files, frames are often dependent of each other and they cannot be cut off just like that.

When you want to read info about an MPEG file, it is usually enough to find the first frame, read its header and assume that the other frames are the same This may not be always the case. Variable bitrate MPEG files may use so called bitrate switching, which means that bitrate changes according to the content of each frame. This way lower bitrates may be used in frames where it will not reduce sound quality. This allows making better compression while keeping high quality of sound.

The frame header is constituted by the very first four bytes (32bits) in a frame. The first eleven bits (or first twelve bits, see below about frame sync) of a frame header are always set and they are called "frame sync". Therefore, you can search through the file for the first occurence of frame sync (meaning that you have to find a byte with a value of 255, and followed by a byte with its three (or four) most significant bits set). Then you read the whole header and check if the values are correct. You will see in the following table the exact meaning of each bit in the header, and which values may be checked for validity. Each value that is specified as reserved, invalid, bad, or not allowed should indicate an invalid header. Remember, this is not enough, frame sync can be easily (and very frequently) found in any binary file. Also it is likely that MPEG file contains garbage on it’s beginning which also may contain false sync. Thus, you have to check two or more frames in a row to assure you are really dealing with MPEG audio file.

Frames may have a CRC check. The CRC is 16 bits long and, if it exists, it follows the frame header. After the CRC comes the audio data. You may calculate the length of the frame and use it if you need to read other headers too or just want to calculate the CRC of the frame, to compare it with the one you read from the file. This is actually a very good method to check the MPEG header validity.

Here is "graphical" presentation of the header content. Characters from A to M are used to indicate different fields. In the table, you can see details about the content of each field.

```
AAAAAAAA AAABBCCD EEEEFFGH IIJJKLMM
```

Sign	Length (bits)	Position (bits)	Description
A	11	(31-21)	Frame sync (all bits set)
B	2	(20,19)	MPEG Audio version ID
			00 - MPEG Version 2.5
			01 - reserved
			10 - MPEG Version 2 (ISO/IEC 13818-3)
			11 - MPEG Version 1 (ISO/IEC 11172-3)
			Note: MPEG Version 2.5 is not official standard. Bit No 20 in frame header is used to indicate version 2.5. Applications that do not support this MPEG version expect this bit always to be set, meaning that frame sync (A) is twelve bits long, not eleve as stated here. Accordingly, B is one bit long (represents only bit No 19). I recommend using methodology presented here, since this allows you to distinguish all three versions and keep full compatibility.
C	2	(18,17)	Layer description
			00 - reserved
			01 - Layer III
			10 - Layer II
			11 - Layer I
D	1	(16)	Protection bit
			0 - Protected by CRC (16bit crc follows header)
			1 - Not protected
E	4	(15,12)	Bitrate index

bits	V1,L1	V1,L2	V1,L3	V2,L1	V2, L2 & L3
0000	free	free	free	free	free
0001	32	32	32	32	8
0010	64	48	40	48	16
0011	96	56	48	56	24
0100	128	64	56	64	32
0101	160	80	64	80	40
0110	192	96	80	96	48
0111	224	112	96	112	56
1000	256	128	112	128	64
1001	288	160	128	144	80
1010	320	192	160	160	96
1011	352	224	192	176	112
1100	384	256	224	192	128
1101	416	320	256	224	144
1110	448	384	320	256	160
1111	bad	bad	bad	bad	bad

NOTES: All values are in kbps
V1 - MPEG Version 1
V2 - MPEG Version 2 and Version 2.5
L1 - Layer I
L2 - Layer II
L3 - Layer III
"free" means free format. If the correct fixed bitrate (such files cannot use variable bitrate) is different than those presented in upper table it must be determined by the application. This may be implemented only for internal purposes since third party applications have no means to find out correct bitrate. However, this is not impossible to do but demands lot's of efforts.
"bad" means that this is not an allowed value

MPEG files may have variable bitrate (VBR). This means that bitrate in the file may change. I have learned about two used methods:
·bitrate switching. Each frame may be created with different bitrate. It may be used in all layers. Layer III decoders must support this method. Layer I & II decoders may support it.
·bit reservoir. Bitrate may be borrowed (within limits) from previous frames in order to provide more bits to demanding parts of the input signal. This causes, however, that the frames are no longer independent, which means you should not cut this files. This is supported only in Layer III.

More about VBR you may find on Xing Tech site

For Layer II there are some combinations of bitrate and mode which are not allowed. Here is a list of allowed combinations.

bitrate	allowed modes
free	all
32	single channel
48	single channel
56	single channel
64	all
80	single channel
96	all
112	all
128	all
160	all
192	all
224	stereo, intensity stereo, dual channel
256	stereo, intensity stereo, dual channel
320	stereo, intensity stereo, dual channel
384	stereo, intensity stereo, dual channel

F 2 (11,10) Sampling rate frequency index (values are in Hz)

bits	MPEG1	MPEG2	MPEG2.5
00	44100	22050	11025
01	48000	24000	12000
10	32000	16000	8000
11	reserv.	reserv.	reserv.

G 1 (9) Padding bit
0 - frame is not padded
1 - frame is padded with one extra slot
Padding is used to fit the bit rates exactly. For an example: 128k 44.1kHz layer II uses a lot of 418 bytes and some of 417 bytes long frames to get the exact 128k bitrate. For Layer I slot is 32 bits long, for Layer II and Layer III slot is 8 bits long.

How to calculate frame length

First, let’s distinguish two terms frame size and frame length. Frame size is the number of samples contained in a frame. It is constant and always 384 samples for Layer I and 1152 samples for Layer II and Layer III. Frame length is length of a frame when compressed. It is calculated in slots. One slot is 4 bytes long for Layer I, and one byte long for Layer II and Layer III. When you are reading MPEG file you must calculate this to be able to find each consecutive frame. Remember, frame length may change from frame to frame due to padding or bitrate switching.

Read the BitRate, SampleRate and Padding of the frame header.

For Layer I files us this formula:

FrameLengthInBytes = (12 * BitRate / SampleRate + Padding) * 4

For Layer II & III files use this formula:

FrameLengthInBytes = 144 * BitRate / SampleRate + Padding

Example:
Layer III, BitRate=128000, SampleRate=441000, Padding=0
==> FrameSize=417 bytes

H 1 (8) Private bit. It may be freely used for specific needs of an application, i.e. if it has to trigger some application specific events.

I 2 (7,6) Channel Mode
00 - Stereo
01 - Joint stereo (Stereo)
10 - Dual channel (Stereo)
11 - Single channel (Mono)

J 2 (5,4) Mode extension (Only if Joint stereo)

Mode extension is used to join informations that are of no use for stereo effect, thus reducing needed resources. These bits are dynamically determined by an encoder in Joint stereo mode.

Complete frequency range of MPEG file is divided in subbands There are 32 subbands. For Layer I & II these two bits determine frequency range (bands) where intensity stereo is applied. For Layer III these two bits determine which type of joint stereo is used (intensity stereo or m/s stereo). Frequency range is determined within decompression algorythm.

Layer I and II		Layer III	
value	Layer I & II	Intensity stereo	MS stereo
00	bands 4 to 31	off	off
01	bands 8 to 31	on	off
10	bands 12 to 31	off	on
11	bands 16 to 31	on	on

K	1	(3)	Copyright 0 - Audio is not copyrighted 1 - Audio is copyrighted
L	1	(2)	Original 0 - Copy of original media 1 - Original media
M	2	(1,0)	Emphasis 00 - none 01 - 50/15 ms 10 - reserved 11 - CCIT J.17

MPEG Audio Tag ID3v1

The TAG is used to describe the MPEG Audio file. It contains information about artist, title, album, publishing year and genre. There is some extra space for comments. It is exactly 128 bytes long and is located at very end of the audio data. You can get it by reading the last 128 bytes of the MPEG audio file.

```
AAABBBBB BBBBBBBB BBBBBBBB BBBBBBBB
BCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCD
DDDDDDDD DDDDDDDD DDDDDDDD DDDDDEEE
EFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFG
```

Sign	Length (bytes)	Position (bytes)	Description
A	3	(0-2)	Tag identification. Must contain ’TAG’ if tag

			exists and is correct.
B	30	(3-32)	Title
C	30	(33-62)	Artist
D	30	(63-92)	Album
E	4	(93-96)	Year
F	30	(97-126)	Comment
G	1	(127)	Genre

The specification asks for all fields to be padded with null character (ASCII 0). However, not all applications respect this (an example is WinAmp which pads fields with <space>, ASCII 32).

There is a small change proposed in **ID3v1.1** structure. The last byte of the Comment field may be used to specify the track number of a song in an album. It should contain a null character (ASCII 0) if the information is unknown.

Genre is a numeric field which may have one of the following values:

0 'Blues'	20 'Alternative'	40 'AlternRock'	60 'Top 40'
1 'Classic Rock'	21 'Ska'	41 'Bass'	61 'Christian Rap'
2 'Country'	22 'Death Metal'	42 'Soul'	62 'Pop/Funk'
3 'Dance'	23 'Pranks'	43 'Punk'	63 'Jungle'
4 'Disco'	24 'Soundtrack'	44 'Space'	64 'Native American'
5 'Funk'	25 'Euro-Techno'	45 'Meditative'	65 'Cabaret'
6 'Grunge'	26 'Ambient'	46 'Instrumental Pop'	66 'New Wave'
7 'Hip-Hop'	27 'Trip-Hop'	47 'Instrumental Rock'	67 'Psychadelic'
8 'Jazz'	28 'Vocal'	48 'Ethnic'	68 'Rave'
9 'Metal'	29 'Jazz+Funk'	49 'Gothic'	69 'Showtunes'
10 'New Age'	30 'Fusion'	50 'Darkwave'	70 'Trailer'
11 'Oldies'	31 'Trance'	51 'Techno-Industrial'	71 'Lo-Fi'
12 'Other'	32 'Classical'	52 'Electronic'	72 'Tribal'
13 'Pop'	33 'Instrumental'	53 'Pop-Folk'	73 'Acid Punk'
14 'R&B'	34 'Acid'	54 'Eurodance'	74 'Acid Jazz'
15 'Rap'	35 'House'	55 'Dream'	75 'Polka'
16 'Reggae'	36 'Game'	56 'Southern Rock'	76 'Retro'
17 'Rock'	37 'Sound Clip'	57 'Comedy'	77 'Musical'
18 'Techno'	38 'Gospel'	58 'Cult'	78 'Rock & Roll'
19 'Industrial'	39 'Noise'	59 'Gangsta'	79 'Hard Rock'

WinAmp expanded this table with next codes:

80	'Folk'	92	'Progressive Rock'	104	'Chamber Music'	116	'Ballad'
81	'Folk-Rock'	93	'Psychedelic Rock'	105	'Sonata'	117	'Poweer Ballad'

82	’National Folk’	94	’Symphonic Rock’	106	’Symphony’	118	’Rhytmic Soul’
83	’Swing’	95	’Slow Rock’	107	’Booty Brass’	119	’Freestyle’
84	’Fast Fusion’	96	’Big Band’	108	’Primus’	120	’Duet’
85	’Bebob’	97	’Chorus’	109	’Porn Groove’	121	’Punk Rock’
86	’Latin’	98	’Easy Listening’	110	’Satire’	122	’Drum Solo’
87	’Revival’	99	’Acoustic’	111	’Slow Jam’	123	’A Capela’
88	’Celtic’	100	’Humour’	112	’Club’	124	’Euro-House’
89	’Bluegrass’	101	’Speech’	113	’Tango’	125	’Dance Hall’
90	’Avantgarde’	102	’Chanson’	114	’Samba’		
91	’Gothic Rock’	103	’Opera’	115	’Folklore’		

Any other value should be considered as ’Unknown’

MPEG Audio Tag ID3v2

This is new proposed TAG format which is different than ID3v1 and ID3v1.1. Complete tech specs for it may be found at http://www.id3.org/.

Created on September 1998. by Predrag Supurovic.
Thanks to Jean for debugging and polishing of this document, Peter Luijer, Guwani, Rob Leslie and Franc Zijderveld for valuable comments and corrections.

Advanced Video Compression - Part 2
Various Codecs, Formats and their Pros and Cons

Continued from the first p

II. Distribution Codecs

1) **MPEG1** - The Golden Oldie

Origins: Back in the very late 80's, the **M**otion **P**icture **E**xperts **G**roup was trying to come up with a open, efficient video compression standard which was officially brought into the International Standa Organization as ISO/IEC-11172, or more commonly known as MPEG1. The MPEG1 standard was designed originally for 1.5MBit/second datarates (i.e. 1X CD-ROM speed, the 1.5 figure includes au however) and 352x240 resolution.

How it Works: MPEG relies heavily on Inter-frame compression. There are 3 different types of fran in the MPEG standard - Intraframes (I-frames), Predicate frames (P-frames), and Bidirectially-interpolated frames (B-frames). I-frames are essentially the MPEG equivalent of Keyframes - these frames have all the data necessary to recreate themselves, i.e. you don't need to reference another frame to render an I-frame. P-frames reference the previous I or P frame and store changes in picture. B-frames are meant to be very low-bitrate frames, so they reference both past and future frames. The standard MPEG stream structure used in VCDs, SVCDs, and DVDs is IBBPBBPBBPBBPBBPBB, although the official MPEG standard doesn't place any restrictions on th distance between I or P frames. A group of frames that starts with an I frame and ends at the frame before the next I frame is called a Group of Pictures, or GOP. It's perfectly legal to stop a GOP short insert a new keyframe and begin a new GOP at any time. This is akin to Variable Keyframe Interval other codecs.

Benefits: MPEG1 was the codec which made distributable digital video happen. MPEG could be streamed, stored on CDs - you could do all sorts of stuff with it. Even today it is still a very viable compression algorithm. You can play MPEG movies on almost anything, from settop players, to PCs handhelds running PocketPC! That's pretty darn impressive!

Disadvantages: MPEG1 is old, lets make no bones about it. It's still damn good, and it beats out everything else in terms of compatibility, but in terms of technical quality there are better things out there. In order to get a good picture, it does require more bits/second than other codecs.

Recommendations: MPEG1, while it won't make the smallest files, is the KING in terms of compatibility. If you want everything under the sun to be able to play your AMVs, distribute in 352x MPEG1 files. Otherwise, I'd stay away from it.

2) **MPEG2** - The Behemoth

Origins: After the MPEG1 standard was finalized, and people started trying to apply it to higher-resolution video pictures, there were many flaws in the standard that became apparent. The biggest of these was that MPEG1 could only compress progressive-scan images, which meant real T pictures (which were interlaced) were very difficult to compress. MPEG2 was born out of the desire achieve compression of broadcast-quality video, and to this end it has succeeded. MPEG2 is used for DVDs, ATSC (High Definition Television) broadcasts, Personal Video Recorders (such as TiVo), an many other applications. MPEG2 is so versatile that while originally it was planned to have an MPE standard for High-Definition TV broadcasts, it turned out that MPEG2 scaled in terms of bitrate so th only 1 standard was necessary for both Standard and High Definition video.

How it Works: MPEG2 is very similar to MPEG1 when you look at the surface, although several of underlying pieces have been completely changed. But for our purposes, it's good enough to say that MPEG2 is a souped-up version of MPEG1 with support for Interlaced video and High-res pictures.

Benefits: MPEG2 gives better picture quality than MPEG1 at full CCIR 601 resolution (720x480) ar at comparable bitrate. Basically, MPEG2 scales, while MPEG1 doesn't.

Disadvantages: MPEG2 takes more CPU horsepower to decompress, and basically has no advantag over MPEG1 at lower resolutions. If you're distributing for the web, you're not going to want to be compressing stuff in MPEG2 instead of MPEG1 because the benefits of MPEG2 aren't realized until you get into full TV resolution, which means big filesizes.

Recommendation: If you're creating your own high-quality archival backups of your videos, I woul encode them at DVD-quality MPEG2, but I would never use MPEG2 for online distribution.

3) **ASF** - **A**dvanced **S**treaming **F**ormat, with the emphasis on STREAMING

Origins: Microsoft, seeing the success people like Apple and RealNetworks were having in the streaming video arena, they decided to barge in, ignore the previous standards and create their own, a Microsoft usually does. Thus ASF was born, a watered down version of AVI paired with an extra information stream. ASF is both a format and a codec. It's basically AVI that can only use Microsoft MPEG4 implementation, but without the indexing issues of AVI (you must have all of an AVI to pla as index blocks and such are referenced at the end of the file).
A newer, standalone/streaming version of ASF called WMV was introduced not too long ago, but it' essentially the same thing, including the drawbacks of data overhead and such.

How it Works: Essentially, ASF is Microsoft's own partial implementation of the MPEG4 standard. say partial because if you read the MPEG4 standard, you will notice that there's a hell of a lot more t the standard than what Microsoft provides. Basically they just took the video encoding portion, stripp out things like B-frames which weren't compatible with the AVI format, and wrote an encoder/decoc for it and called it Windows Media.

Benefits: If you want to stream your video over the web, ASF may be very attractive to you. It's a format which was meant for streaming, so most of its architecture is centered around that fact. Howe I doubt most of you have the bandwidth to stream your videos live over the web.

Disadvantages: Just as it's an advantage for streaming, the structure of ASF is also very prohibitive when you're trying to simply store video in single files, to be downloaded and viewed later. ASF kee track of all sorts of other things in its stream, meaning there's significant bitrate overhead for non-vic and non-audio data (example, using the old VirtualDub 1.3c and the 4.0 beta of Windows Media Too

where the MP43 codec still worked in AVIs, an average ASF file takes up 25% more space than a di stream copy of the video stream to an AVI file). ASF is limited to 352x288 resolution. ASF is also p to errors, as when you compress to ASF the encoder does things like dropping frames to maintain bitrate, and it also does some other nasty things which when streaming isn't an issue, but definitely i when you're just trying to encode standalone video files.
As if that all weren't bad enough, Microsoft is also very protective of its format. They've patented th ASF file structure so that no program can interface with it without breaking patent law, and their lice agreements (or at least they used to) actually forbid transcoding to other formats after encoding to A

Recommendation: Don't use it unless you're planning on streaming your files.

4) **DivX ;-) 3.11a Alpha** - ASF without the drawbacks

Origins: A guy who went by the handle "Gej" used a hex editor and some other tools to modify the binary dll file of Microsoft's MS-MPEG4v3 ASF codec so that it would work under AVIs, so that yo could use it to encode high-res movies (since ASF was restricted to 352x288 resolution). He then released it onto the net. This became the original DivX which was updated until version 3.11alpha.

How it Works: Essentially it's the same codec as what's used in ASF (if you don't believe me, use t ASF VirtualDub to extract an ASF video stream to an AVI file, then change the AVI's FourCC code DIV3 and notice it still works) except that it uses the AVI file format, meaning no resolution restricti no filesize overhead, and easy manipulation and playback in any program supporting AVI.

Benefits: DivX, like ASF, is a decent MPEG4 implementation (although incomplete). It can compre images better than MPEG1 in less bits, but it does lose more color data than MPEG1 (at least in Microsoft's implementation it does). However it's since been surpassed in quality by XviD but still h better compatibility.

Disadvantages: DivX isn't quite as compatible as MPEG1 - it plays on PCs, BeOS, and Linux easil enough, and it will play on the Mac with some work, but on other platforms you're out of luck. It als takes more horsepower to playback than MPEG1 (not as much as MPEG2 though), but there are setti in the advanced area of the Codec for playback on slower machines at slightly lower video quality. D also has some issues with solid black not being black.

Recommendation: At this time I would recommend encoding to XviD if you want MPEG4 encodin but DivX3 is still a very viable codec and when used properly with NanDub it can make very good encodes.

5) **DivX4/5** - The sequel to DivX

Origins: After the success of the original DivX3 codec, a group got together and started an open sou version of DivX from scratch called OpenDivX. Unfortunately, they later closed the source and start working on it solely in house. First they released DivX4, which was alright but wasn't as good as DivX3. With the release of DivX5, they managed to hack in B-frames into an AVI file, which is a pr impressive feat, although they way they do it is very messy and produces AVI files with rather bizar structure, but they'll playback via DirectShow just fine.

How it Works: DivX5 is a pretty standard MPEG4 codec, and in fact can produce MPEG4 complia streams if you encode audio to AAC and distribute as .mp4 files. However you'll need an MPEG4 software player or wait until Microsoft includes MPEG4 playback in their media player.

Benefits: DivX5 is pretty easy to setup and work with, but it doesn't have very many other advantag for encoding. It's playback filter, though, is fast and offers good post-processing capability. The playback filter's also easy to install and works good on properly encoded XviD files so we actually recommend installing DivX5 even if you don't plan on encoding with it. DivX5 does have very good playback options for PC, Linux, and the Mac, which covers the three biggest desktop platforms.

Disadvantages: DivX5's quality, while pretty good, still doesn't surpass the original DivX3 in my opinion, and the inability to really tweak the codec's internal options are a big downside. Also, while B-frames do improve compressibility, they look very bad and so I wouldn't use them anyway. Also, order to get all the options you have to either pay money for it, or download a version with adware ar spyware attached to it. While it's possible to remove it, I think it's insulting and I don't want to get it near my machine. Fortunately you can download a full-fledged decoder in the non-adware version.

Recommendation: I don't recommend encoding your movies in DivX5 due to its questionable quali and adware. DivX3 as well as XviD are better options for MPEG4 encoding, IMO

6) XviD - Fast, Open Source, Customizable, and Gorgeous

Origins: When OpenDivX closed its source and went on to produce DivX4, a group of open source developers decided to take the code that was left from OpenDivX and start their own codec. Thus Xv (or DivX backwards) was born. By now very little of the original OpenDivX code remains but there' few bits here and there.

How it Works: XviD is an MPEG4 compliant codec which performs very well and has oodles of options. While it doesn't do B-frames yet (and when they do they may not be compatible with the DivX5 playback filter) it does implement many of the options of MPEG4 and very well, at that.

Benefits: XviD has all sorts of tweakable options, which means an experienced encoder (or one following a well-written guide) can produce some very nice looking encodes. It can also produce MPEG4 compatible streams if you want to distribute as .MP4 files. Usually the latest snapshots relea by Koepi are pretty stable and a good representation of the state of the code, and since XviD's alway producing MPEG4 compatible streams, you should never run into a situation where your old encodes no longer compatible. The new versions will just do some things better or add new features. Also utilizing some of the programs provided by DivX Networks, you can get XviD files to playback on t same platforms DivX can, which includes PC, Linux, and Mac.

Disadvantages: Unlike DivX5, it can be difficult to setup and get something that looks good. It also still in its alpha stages of development, which means there may be bugs. XviD also lacks it's own go DirectShow filter. The XviD specific filter has very few good post-processing capabilities, and while ffdshow will do decent decoding, it's not very user-friendly and changes too often to be a good thing recommend your viewers install. Fortunately you can encode your XviD files to playback correctly w the DivX5 playback filter.

Recommendation: This is what I use exclusively now to distribute my files. It provides excellent quality and compressibility as well as advanced features, and it has good cross-compatability among desktop operating systems.

7) **Quicktime** - Apple’s video platform

No, Quicktime isn’t a codec in and of itself - it’s more of a format like AVI is. However, I recommen against distributing your videos as any Quicktime format because of the requirements of its player. While not as intrusive as say RealPlayer, Quicktime’s player for Windows is slow, clunky, and unsta Also QT playback in Linux is sketchy at best. While Apple claims to have a fully compliant MPEG4 codec in their Quicktime Version 6 package, tests have shown that it does not play back real MPEG4 streams, nor will compliant players playback it’s MPEG4 streams. Consider QT’s MPEG4 codec to b the same as Microsoft’s MPEG4 codec - they’re open standards, yet proprietary in practice.

ErMaC - September 2

Next Time - Getting Your Video

Index

Audio-Burner.com

DVD Formats Explained

DVD which in the past has been called Digital Video Disc, but is more commonly referred to as Digital Versatile Disc is one of the fastest growing consumer electronic products in history. With that are a number of competing formats looking to become the de-facto standard, the way that CD-R/W has become in the computer industry.

In due time, as formats are standardized, inexpensive DVD burners will become as common as CD burners and along with that will be the availability of affordable DVD software and DVD blank media.

The method of using your DVD burner on your computer will be no different than what you are currently used to with your CD-R/W burners and CD burning software. It's just a matter of patience and time before the industry sorts things out because DVD burners are set to take off the same way as CD burners did a few years ago. Let's now take a look at the various DVD formats available today.

Format	**Description**

Acoustica MP3 Software

:: MP3 CD Burner
:: MP3 to Wave Converter
:: MP3 Audio Mixer
:: CD Label Maker

MP3 Converter Software

:: Audio Convert
:: Advanced WMA Workshop

MP3 CD Burner Software

:: Data & MP3 CD Burner
:: MP3 CD Converter
:: X2CD Music CD Burner

CD Ripper Software

DVD Audio	DVD Audio provides higher-quality audio than available from current CDs. DVD Audio offers higher quality audio including Dolby Digital AC-3 and surround sound, and a wide range of options for coding audio at high fidelity, with 24 bits per sample and 96 KHz sampling frequency and beyond. In addition, look for features such as still pictures, text information, menus and navigation, and even video sequences. The format provides for longer playing times; a dual layer DVD Audio disc will hold at least 2 hours of full surround sound audio. For the recording industry, DVD Audio includes copy protection and anti-piracy measures. Consumer response has been slow and DVD Audio shouldn't displace CD audio as the standard any time soon.
DVD Video	This is the format used by Hollywood and by consumers for viewing movies and other visual entertainment. The total capacity is 17 gigabytes if two layers on both sides of the disk are used.
DVD-ROM	Its basic technology is the same as DVD Video, but it also includes computer friendly file formats which be used to store data. This product should replace conventional CD-ROMs over time.

CD Ripper Software

:: MP3 CD Extractor
:: AltoMP3 Maker
:: MP3 Master

MP3 Editors and Sound Recorders

:: Coding Workshop Ringtone Converter

DVD Software

:: DVD X Copy!
:: DVD Squeeze
:: DVD Echo
:: Super DVD Ripper

DVD-RAM	Think of a DVD-RAM as a virtual hard disk, with a random read-write access. Originally a 2.6GB drive, its capacity has increased to 4.7GB per side. Double sided DVD-RAM media is now available with a 9.4GB capacity and can be re-written more than 100,000 times and does not need to be reformatted when you want to re-write. You can drag and drop files to a DVD-RAM drive as if it were a regular hard drive. However, DVD-RAM disks can not be played in existing DVD players and DVD-ROM drives. You will require a DVD-RAM drive to playback DVD-RAMs.
DVD-R	Developed by Pioneer, DVD-R, with a capacity of 4.7GB per side is similar to a DVD-ROM but allows users to write only once. Originally designed for professional authoring DVD-R(A), a version for general consumer use is now available DVD-R(G). The major difference between professional and general authoring is that professional supports Mastering and Copy Protection. DVD-R disks can be played in most DVD players and DVD-ROM drives
DVD-RW	DVD-RW is an extension of the DVD-R format with a read-write capacity of 4.7GB per side. It can be re-written up to about 1,000 times. Like DVD-R, DVD-RW disks can be played back in most DVD players and DVD-ROM drives

DVD+RW	Developed in co-operation by Hewlett-Packard, Mitsubishi Chemical, Philips, Ricoh, Sony, Dell, Compaq and Yamaha, DVD+RW is the only re-writable format that provides full compatibility with existing DVD-Video players and DVD-ROM drives. Does not read or write DVD-RAM discs but will continue to write CD-Rs and CD-RWs. This technology is based on the CD-R/RW format and has a read-write capacity of 4.7GB per side which can be re-written up to 1,000 times. A single write version of this technology called DVD+R is expected in 2002.

Confused? Don’t worry about it because even the most seasoned professional is trying to get their heads around this. In due time, a standard will evolve and it will be easier to understand and use the technology. If you are interested in learning more, visit the DVD+R/W Alliance or DVD Forum to stay informed about the latest industry news.

Appendices

Here are some more detailed pieces of info that I received by e-mail.
They are reproduced here virtually without much editing.

Table of contents

FTP access for non-internet sites

From the sci.space FAQ:

> Sites not connected to the Internet cannot use FTP directly, but
> there are a few automated FTP servers which operate via email.
> Send mail containing only the word HELP to ftpmail@decwrl.dec.com
> or bitftp@pucc.princeton.edu, and the servers will send you
> instructions on how to make requests. (The bitftp service is no
> longer available through UUCP gateways due to complaints about
> overuse :-( )

Also:

> FAQ lists are available by anonymous FTP from rftm.mit.edu
> and by email from mail-server@rtfm.mit.edu (send a message
> containing "help" for instructions about the mail server).

AIFF Format (Audio IFF) and AIFC

This format was developed by Apple for storing high-quality sampled
sound and musical instrument info; it is also used by SGI and several
professional audio packages (sorry, I know no names). An extension,
called AIFC or AIFF-C, supports compression (see the last item below).

I’ve made a BinHex’ed MacWrite version of the AIFF spec (no idea if
it’s the same text as mentioned below) available by anonymous ftp from
ftp.cwi.nl [192.16.184.180]; the file is /pub/audio/AudioIFF1.2.hqx.
But you may be better off with the AIFF-C specs, see below.

Mike Brindley (brindley@ece.orst.edu) writes:

"The complete AIFF spec by Steve Milne, Matt Deatherage (Apple) is

available in ’AMIGA ROM Kernal Reference Manual: Devices (3rd Edition)’
1991 by Commodore-Amiga, Inc.; Addison-Wesley Publishing Co.;
ISBN 0-201-56775-X, starting on page 435 (this edition has a charcoal
grey cover). It is available in most bookstores, and soon in many
good librairies."

According to Mark Callow (msc@sgi.com):

A PostScript version of the AIFF-C specification is available via
anonymous ftp on FTP.SGI.COM (192.48.153.1) as /sgi/aiff-c.9.26.91.ps.

The NeXT/Sun audio file format

Here’s the complete story on the file format, from the NeXT
documentation. (Note that the "magic" number is ((int)0x2e736e64),
which equals ".snd".) Also, at the end, I’ve added a litte document
that someone posted to the net a couple of years ago, that describes
the format in a bit-by-bit fashion rather than from C.

I received this from Doug Keislar, NeXT Computer. This is also the
Sun format, except that Sun doesn’t recognize as many format codes. I
added the numeric codes to the table of formats and sorted it.

SNDSoundStruct: How a NeXT Computer Represents Sound

The NeXT sound software defines the SNDSoundStruct structure to
represent sound. This structure defines the soundfile and Mach-O
sound segment formats and the sound pasteboard type. It’s also used
to describe sounds in Interface Builder. In addition, each instance
of the Sound Kit’s Sound class encapsulates a SNDSoundStruct and
provides methods to access and modify its attributes.

Basic sound operations, such as playing, recording, and cut-and-paste
editing, are most easily performed by a Sound object. In many cases,
the Sound Kit obviates the need for in-depth understanding of the
SNDSoundStruct architecture. For example, if you simply want to
incorporate sound effects into an application, or to provide a simple
graphic sound editor (such as the one in the Mail application), you
needn’t be aware of the details of the SNDSoundStruct. However, if
you want to closely examine or manipulate sound data you should be
familiar with this structure.

The SNDSoundStruct contains a header, information that describes the
attributes of a sound, followed by the data (usually samples) that
represents the sound. The structure is defined (in
sound/soundstruct.h) as:

```
typedef struct {
    int magic;               /* magic number SND_MAGIC */
    int dataLocation;        /* offset or pointer to the data */
    int dataSize;            /* number of bytes of data */
    int dataFormat;          /* the data format code */
    int samplingRate;        /* the sampling rate */
    int channelCount;        /* the number of channels */
    char info[4];            /* optional text information */
} SNDSoundStruct;
```

SNDSoundStruct Fields

magic

magic is a magic number that’s used to identify the structure as a SNDSoundStruct. Keep in mind that the structure also defines the soundfile and Mach-O sound segment formats, so the magic number is also used to identify these entities as containing a sound.

dataLocation

It was mentioned above that the SNDSoundStruct contains a header followed by sound data. In reality, the structure only contains the header; the data itself is external to, although usually contiguous with, the structure. (Nonetheless, it’s often useful to speak of the SNDSoundStruct as the header and the data.) dataLocation is used to point to the data. Usually, this value is an offset (in bytes) from the beginning of the SNDSoundStruct to the first byte of sound data. The data, in this case, immediately follows the structure, so dataLocation can also be thought of as the size of the structure’s header. The other use of dataLocation, as an address that locates data that isn’t contiguous with the structure, is described in "Format Codes," below.

dataSize, dataFormat, samplingRate, and channelCount

These fields describe the sound data.

dataSize is its size in bytes (not including the size of the SNDSoundStruct).

dataFormat is a code that identifies the type of sound. For sampled sounds, this is the quantization format. However, the data can also be instructions for synthesizing a sound on the DSP. The codes are listed and explained in "Format Codes," below.

samplingRate is the sampling rate (if the data is samples). Three sampling rates, represented as integer constants, are supported by the hardware:

Constant	Sampling Rate (samples/sec)
SND_RATE_CODEC	8012.821 (CODEC input)
SND_RATE_LOW	22050.0 (low sampling rate output)
SND_RATE_HIGH	44100.0 (high sampling rate output)

channelCount is the number of channels of sampled sound.

info

info is a NULL-terminated string that you can supply to provide a textual description of the sound. The size of the info field is set when the structure is created and thereafter can’t be enlarged. It’s at least four bytes long (even if it’s unused).

Format Codes

A sound’s format is represented as a positive 32-bit integer. NeXT reserves the integers 0 through 255; you can define your own format and represent it with an integer greater than 255. Most of the formats defined by NeXT describe the amplitude quantization of sampled sound data:

Value	Code	Format
0	SND_FORMAT_UNSPECIFIED	unspecified format
1	SND_FORMAT_MULAW_8	8-bit mu-law samples
2	SND_FORMAT_LINEAR_8	8-bit linear samples
3	SND_FORMAT_LINEAR_16	16-bit linear samples
4	SND_FORMAT_LINEAR_24	24-bit linear samples
5	SND_FORMAT_LINEAR_32	32-bit linear samples
6	SND_FORMAT_FLOAT	floating-point samples
7	SND_FORMAT_DOUBLE	double-precision float samples
8	SND_FORMAT_INDIRECT	fragmented sampled data
9	SND_FORMAT_NESTED	?
10	SND_FORMAT_DSP_CORE	DSP program
11	SND_FORMAT_DSP_DATA_8	8-bit fixed-point samples
12	SND_FORMAT_DSP_DATA_16	16-bit fixed-point samples
13	SND_FORMAT_DSP_DATA_24	24-bit fixed-point samples
14	SND_FORMAT_DSP_DATA_32	32-bit fixed-point samples
15	?	
16	SND_FORMAT_DISPLAY	non-audio display data
17	SND_FORMAT_MULAW_SQUELCH	?
18	SND_FORMAT_EMPHASIZED	16-bit linear with emphasis
19	SND_FORMAT_COMPRESSED	16-bit linear with compression
20	SND_FORMAT_COMPRESSED_EMPHASIZED	A combination of the two above
21	SND_FORMAT_DSP_COMMANDS	Music Kit DSP commands
22	SND_FORMAT_DSP_COMMANDS_SAMPLES	?
[Some new ones supported by Sun. This is all I currently know. --GvR]		
23	SND_FORMAT_ADPCM_G721	
24	SND_FORMAT_ADPCM_G722	
25	SND_FORMAT_ADPCM_G723_3	
26	SND_FORMAT_ADPCM_G723_5	
27	SND_FORMAT_ALAW_8	

Most formats identify different sizes and types of sampled data. Some deserve special note:

-- SND_FORMAT_DSP_CORE format contains data that represents a loadable DSP core program. Sounds in this format are required by the SNDBootDSP() and SNDRunDSP() functions. You create a SND_FORMAT_DSP_CORE sound by reading a DSP load file (extension

".lod") with the SNDReadDSPfile() function.

--   SND_FORMAT_DSP_COMMANDS is used to distinguish sounds that contain DSP commands created by the Music Kit. Sounds in this format can only be created through the Music Kit's Orchestra class, but can be played back through the SNDStartPlaying() function.

--   SND_FORMAT_DISPLAY format is used by the Sound Kit's SoundView class. Such sounds can't be played.

--   SND_FORMAT_INDIRECT indicates data that has become fragmented, as described in a separate section, below.

--   SND_FORMAT_UNSPECIFIED is used for unrecognized formats.

Fragmented Sound Data

Sound data is usually stored in a contiguous block of memory. However, when sampled sound data is edited (such that a portion of the sound is deleted or a portion inserted), the data may become discontiguous, or fragmented. Each fragment of data is given its own SNDSoundStruct header; thus, each fragment becomes a separate SNDSoundStruct structure. The addresses of these new structures are collected into a contiguous, NULL-terminated block; the dataLocation field of the original SNDSoundStruct is set to the address of this block, while the original format, sampling rate, and channel count are copied into the new SNDSoundStructs.

Fragmentation serves one purpose: It avoids the high cost of moving data when the sound is edited. Playback of a fragmented sound is transparent-you never need to know whether the sound is fragmented before playing it. However, playback of a heavily fragmented sound is less efficient than that of a contiguous sound. The SNDCompactSamples() C function can be used to compact fragmented sound data.

Sampled sound data is naturally unfragmented. A sound that's freshly recorded or retrieved from a soundfile, the Mach-O segment, or the pasteboard won't be fragmented. Keep in mind that only sampled data can become fragmented.

____

```
>From mentor.cc.purdue.edu!purdue!decwrl!ucbvax!ziploc!eps Wed Apr  4
23:56:23 EST 1990
Article 5779 of comp.sys.next:
Path: mentor.cc.purdue.edu!purdue!decwrl!ucbvax!ziploc!eps
>From: eps@toaster.SFSU.EDU (Eric P. Scott)
~Newsgroups: comp.sys.next
~Subject: Re: Format of NeXT sndfile headers?
Message-ID: <445@toaster.SFSU.EDU>
~Date: 31 Mar 90 21:36:17 GMT
~References: <14978@phoenix.Princeton.EDU>
```

```
~Reply-To: eps@cs.SFSU.EDU (Eric P. Scott)
Organization: San Francisco State University
~Lines: 42

In article <14978@phoenix.Princeton.EDU>
        bskendig@phoenix.Princeton.EDU (Brian Kendig) writes:
>I'd like to take a program I have that converts Macintosh sound
files
>to NeXT sndfiles and polish it up a bit to go the other direction as
>well.

Two people have already submitted programs that do this
(Christopher Lane and Robert Hood); check the various
NeXT archive sites.

>       Could someone please give me the format of a NeXT sndfile
>header?

"big-endian"
        0       1       2       3
        +-------+-------+-------+-------+
0       | 0x2e  | 0x73  | 0x6e  | 0x64  |       "magic" number
        +-------+-------+-------+-------+
4       |                               |       data location
        +-------+-------+-------+-------+
8       |                               |       data size
        +-------+-------+-------+-------+
12      |                               |       data format (enum)
        +-------+-------+-------+-------+
16      |                               |       sampling rate (int)
        +-------+-------+-------+-------+
20      |                               |       channel count
        +-------+-------+-------+-------+
24      |       |       |       |       |       (optional) info
string

28 = minimum value for data location

data format values can be found in /usr/include/sound/soundstruct.h

Most common combinations:

         sampling  channel    data
             rate    count  format
voice file   8012        1       1 =  8-bit mu-law
system beep 22050        2       3 = 16-bit linear
CD-quality  44100        2       3 = 16-bit linear

------------------------------------------------------------------------
IFF/8SVX Format
---------------

~Newsgroups: alt.binaries.sounds.d,alt.sex.sounds
~Subject: Format of the IFF header (Amiga sounds)
Message-ID: <2509@tardis.Tymnet.COM>
~From: jms@tardis.Tymnet.COM (Joe Smith)
~Date: 23 Oct 91 23:54:38 GMT
Followup-To: alt.binaries.sounds.d
Organization: BT North America (Tymnet)

The first 12 bytes of an IFF file are used to distinguish between an Amiga
```

picture (FORM-ILBM), an Amiga sound sample (FORM-8SVX), or other file
conforming to the IFF specification. The middle 4 bytes is the count of
bytes that follow the "FORM" and byte count longwords. (Numbers are stored
in M68000 form, high order byte first.)

FutureSound audio file, 15000 samples at 10.000KHz, file is 15048 bytes long.

```
0000: 464F524D 00003AC0 38535658 56484452    FORM..:.8SVXVHDR
      F O R M     15040 8 S V X  V H D R
0010: 00000014 00003A98 00000000 00000000    ......:.........
            20    15000        0        0
0020: 27100100 00010000 424F4459 00003A98    '.......BODY..:.
     10000 1 0    1.0   B O D Y     15000
```

```
0000000..03 = "FORM", identifies this as an IFF format file.
FORM+00..03 (ULONG) = number of bytes that follow.  (Unsigned long int.)
FORM+03..07 = "8SVX", identifies this as an 8-bit sampled voice.

????+00..03 = "VHDR", Voice8Header, describes the parameters for the BODY.
VHDR+00..03 (ULONG) = number of bytes to follow.
VHDR+04..07 (ULONG) = samples in the high octave 1-shot part.
VHDR+08..0B (ULONG) = samples in the high octave repeat part.
VHDR+0C..0F (ULONG) = samples per cycle in high octave (if repeating), else 0.
VHDR+10..11 (UWORD) = samples per second.  (Unsigned 16-bit quantity.)
VHDR+12     (UBYTE) = number of octaves of waveforms in sample.
VHDR+13     (UBYTE) = data compression (0=none, 1=Fibonacci-delta encoding).
VHDR+14..17 (FIXED) = volume.  (The number 65536 means 1.0 or full volume.)

????+00..03 = "BODY", identifies the start of the audio data.
BODY+00..03 (ULONG) = number of bytes to follow.
BODY+04..NNNNN      = Data, signed bytes, from -128 to +127.
```

```
0030: 04030201 02030303 04050605 05060605
0040: 06080806 07060505 04020202 01FF0000
0050: 00000000 FF00FFFF FFFEFDFD FDFEFFFF
0060: FDFDFF00 00FFFFFF 00000000 00FFFF00
0070: 00000000 00FF0000 00FFFEFF 00000000
0080: 00010000 000101FF FF0000FE FEFFFFFE
0090: FDFDFEFD FDFFFFFC FDFEFDFD FEFFFEFE
00A0: FFFEFEFE FEFEFEFF FFFFFEFF 00FFFF01
```

This small section of the audio sample shows the number ranging from -5 (0xFD)
to +8 (0x08). Warning: Do not assume that the BODY starts 48 bytes into the
file. In addition to "VHDR", chunks labeled "NAME", "AUTH", "ANNO", or
"(c) " may be present, and may be in any order. You will have to check the
byte count in each chunk to determine how many bytes to skip.

Playing sound on a PC

~From: Eric A Rasmussen

Any turbo PC (8088 at 8 Mhz or greater)/286/386/486/etc. can produce a quality
playback of single channel 8 bit sounds on the internal (1 bit, 1 channel)
speaker by utilizing Pulse-Width-Modulation, which toggles the speaker faster
than it can physically move to simulate positions between fully on and fully
off. There are several PD programs of this nature that I know of:

```
REMAC  - Plays MAC format sound files.  Files on the Macintosh, at least the
         sound files that I've ripped apart, seem to contain 3 parts.  The
         first two are info like what the file icon looks like and other
         header type info.  The third part contains the raw sample data, and
         it is this portion of the file which is saved to a seperate file,
         often named with the .snd extension by PC users.  Personally, I like
         to name the files .s1, .s2, .s3, or .s4 to indicate the sampling rate
         of the file. (-s# is how to specify the playback rate in REMAC.)
         REMAC provides playback rates of 5550hz, 7333hz, 11 khz, & 22 khz.
REMAC2 - Same as REMAC, but sounds better on higher speed machines.
REPLAY - Basically same as REMAC, but for playback of Atari ST sounds.
         Apparently, the Atari has two sound formats, one of which sounds like
         garbage if played by REMAC or REPLAY in the incorrect mode.  The
         other file format works fine with REMAC and so appears to be 'normal'
         unsigned 8-bit data.  REPLAY provides playback rates of 11.5 khz,
         12.5 khz, 14 khz, 16 khz, 18.5 khz, 22khz, & 27 khz.
```

These three programs are all by the same author, Richard E. Zobell who does not have an internet mail address to my knowledge, but does have a GEnie email address of R.ZOBELL.

Additionally, there are various stand-alone demos which use the internal speaker, of which there is one called mushroom which plays a 30 second advertising jingle for magic mushroom room deoderizers which is pretty humerous. I've used this player to playback samples that I ripped out of the commercial game program Mean Streets, which uses something they call RealSound (tm) to playback digital samples on the internal speaker. (Of course, I only do this on my own system, and since I own the game, I see no problems with it.)

For owners of 8 Mhz 286's and above, the option to play 4 channel 8 bit sounds (with decent quality) on the internal speaker is also a reality. Quite a number of PD programs exist to do this, including, but not limited to:

ModEdit, ModPlay, ScreamTracker, STM, Star Trekker, Tetra, and probably a few more.

All these programs basically make use of various sound formats used by the Amiga line of computers. These include .stm files, .mod files [a.k.a. mod. files], and .nst files [really the same hing]. Also, these programs pretty much all have the option to playback the sound to add-on hardware such as the SoundBlaster card, the Covox series of devices, and also to direct the data to either one or two (for stereo) parallel ports, which you could attach your own D/A's to. (From what I have seen, the Covox is basically an small amplified speaker with a D/A which plugs into the parallel port. This sounds very similiar to the Disney Sound System (DSS) which people have been talking about recently.)

The EA-IFF-85 documentation

~From: dgc3@midway.uchicago.edu

As promised, here's an ftp location for the EA-IFF-85 documentation. It's the November 1988 release as revised by Commodore (the last public release), with specifications for IFF FORMs for graphics, sound, formatted text, and more. IFF FORMS now exist for other media, including structured drawing, and new documentation is now available only from Commodore.

The documentation is at grind.isca.uiowa.edu [128.255.19.233], in the directory /amiga/f1/ff185. The complete file list is as follows:

DOCUMENTS.zoo
EXAMPLES.zoo
EXECUTABLE.zoo
INCLUDE.zoo
LINKER_INFO.zoo
OBJECT.zoo
SOURCE.zoo
TP_IFF_Specs.zoo

All files except DOCUMENTS.zoo are Amiga-specific, but may be used as a basis for conversion to other platforms. Well, I take that tentatively back. I don't know what TP_IFF_Specs.zoo contains, so it might be non-Amiga-specific.

US Federal Standard 1016 availability

~From: jpcampb@afterlife.ncsc.mil (Joe Campbell)

The U.S. DoD's Federal-Standard-1016 based 4800 bps code excited linear prediction voice coder version 3.2 (CELP 3.2) Fortran and C simulation source codes are available for worldwide distribution (on DOS diskettes, but configured to compile on Sun SPARC stations) from NTIS and DTIC. Example input and processed speech files are included. A Technical Information Bulletin (TIB), "Details to Assist in Implementation of Federal Standard 1016 CELP," and the official standard, "Federal Standard 1016, Telecommunications: Analog to Digital Conversion of Radio Voice by 4,800 bit/second Code Excited Linear Prediction (CELP)," are also available.

This is available through the National Technical Information Service:

NTIS
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
USA
(703) 487-4650

The "AD" ordering number for the CELP software is AD M000 118 (US$ 90.00) and for the TIB it's AD A256 629 (US$ 17.50). The LPC-10 standard, described below, is FIPS Pub 137 (US$ 12.50). There is a $3.00 shipping charge on all U.S. orders. The telephone number for their automated system is 703-487-4650, or 703-487-4600 if you'd prefer to talk with a real person.

(U.S. DoD personnel and contractors can receive the package from the Defense Technical Information Center: DTIC, Building 5, Cameron Station, Alexandria, VA 22304-6145. Their telephone number is 703-274-7633.)

The following articles describe the Federal-Standard-1016 4.8-kbps CELP coder (it's unnecessary to read more than one):

Campbell, Joseph P. Jr., Thomas E. Tremain and Vanoy C. Welch, "The Federal Standard 1016 4800 bps CELP Voice Coder," Digital Signal Processing, Academic Press, 1991, Vol. 1, No. 3, p. 145-155.

Campbell, Joseph P. Jr., Thomas E. Tremain and Vanoy C. Welch, "The DoD 4.8 kbps Standard (Proposed Federal Standard 1016)," in Advances in Speech Coding, ed. Atal, Cuperman and Gersho, Kluwer Academic Publishers, 1991, Chapter 12, p. 121-133.

Campbell, Joseph P. Jr., Thomas E. Tremain and Vanoy C. Welch, "The
Proposed Federal Standard 1016 4800 bps Voice Coder: CELP," Speech
Technology Magazine, April/May 1990, p. 58-64.

The U.S. DoD's Federal-Standard-1015/NATO-STANAG-4198 based 2400 bps
linear prediction coder (LPC-10) was republished as a Federal
Information Processing Standards Publication 137 (FIPS Pub 137).
It is described in:

Thomas E. Tremain, "The Government Standard Linear Predictive Coding
Algorithm: LPC-10," Speech Technology Magazine, April 1982, p. 40-49.

There is also a section about FS-1015 in the book:
Panos E. Papamichalis, Practical Approaches to Speech Coding,
Prentice-Hall, 1987.

The voicing classifier used in the enhanced LPC-10 (LPC-10e) is described in:
Campbell, Joseph P., Jr. and T. E. Tremain, "Voiced/Unvoiced Classification
of Speech with Applications to the U.S. Government LPC-10E Algorithm,"
Proceedings of the IEEE International Conference on Acoustics, Speech, and
Signal Processing, 1986, p. 473-6.

Copies of the official standard
"Federal Standard 1016, Telecommunications: Analog to Digital Conversion
of Radio Voice by 4,800 bit/second Code Excited Linear Prediction (CELP)"
are available for US$ 5.00 each from:

GSA Federal Supply Service Bureau
Specification Section, Suite 8100
470 E. L'Enfant Place, S.W.
Washington, DC 20407
(202)755-0325

Realtime DSP code for FS-1015 and FS-1016 is sold by:

John DellaMorte
DSP Software Engineering
165 Middlesex Tpk, Suite 206
Bedford, MA 01730
USA
1-617-275-3733
1-617-275-4323 (fax)
dspse.bedford@channel1.com

DSP Software Engineering's FS-1016 code can run on a DSP Research's Tiger 30
(a PC board with a TMS320C3x and analog interface suited to development work).

DSP Research
1095 E. Duane Ave.
Sunnyvale, CA 94086
USA
(408)773-1042
(408)736-3451 (fax)

~From: tobiasr@monolith.lrmsc.loral.com (Richard Tobias)

For U.S. FED-STD-1016 (4800 bps CELP) _realtime_ DSP code and
information about products using this code using the AT&T DSP32C and

AT&T DSP3210, contact:

White Eagle Systems Technology, Inc.
1123 Queensbridge Way
San Jose, CA 95120
(408) 997-2706
(408) 997-3584 (fax)
rjjt@netcom.com

~From: Cole Erskine <cole@analogical.com>

[paraphrased]

Analogical Systems has a _real-time_ multirate implementation of U.S. Federal Standard 1016 CELP operating at bit rates of 4800, 7200, and 9600 bps on a single 27MHz Motorola DSP56001. Source and object code is available for a one-time license fee.

FREE, _real-time_ demonstration software for the Ariel PC-56D is available for those who already have such a board by contacting Analogical Systems. The demo software allows you to record and playback CELP files to and from the PC’s hard disk.

Analogical Systems
2916 Ramona Street
Palo Alto, CA 94306
Tel: +1 (415) 323-3232
FAX: +1 (415) 323-4222

Creative Voice (VOC) file format

~From: galt@dsd.es.com

(byte numbers are hex!)

```
    HEADER (bytes 00-19)
    Series of DATA BLOCKS (bytes 1A+) [Must end w/ Terminator Block]
```

- --

HEADER:
=======

byte #	Description
00-12	"Creative Voice File"
13	1A (eof to abort printing of file)
14-15	Offset of first datablock in .voc file (std 1A 00 in Intel Notation)
16-17	Version number (minor,major) (VOC-HDR puts 0A 01)
18-19	2’s Comp of Ver. # + 1234h (VOC-HDR puts 29 11)

- --

DATA BLOCK:
===========

```
   Data Block:  TYPE(1-byte), SIZE(3-bytes), INFO(0+ bytes)
   NOTE: Terminator Block is an exception -- it has only the TYPE byte.
```

TYPE	Description	Size (3-byte int)	Info
00	Terminator	(NONE)	(NONE)
01	Sound data	2+length of data	*
02	Sound continue	length of data	Voice Data
03	Silence	3	**
04	Marker	2	Marker# (2 bytes)
05	ASCII	length of string	null terminated string
06	Repeat	2	Count# (2 bytes)
07	End repeat	0	(NONE)
08	Extended	4	***

*Sound Info Format:

00	Sample Rate
01	Compression Type
02+	Voice Data

**Silence Info Format:

00-01	Length of silence - 1
02	Sample Rate

***Extended Info Format:

00-01	Time Constant: Mono: 65536 - (256000000/sample_rate)
	Stereo: 65536 - (25600000/(2*sample_rate))
02	Pack
03	Mode: 0 = mono
	1 = stereo

Marker#	-- Driver keeps the most recent marker in a status byte
Count#	-- Number of repetitions + 1
	Count# may be 1 to FFFE for 0 - FFFD repetitions
	or FFFF for endless repetitions
Sample Rate	-- SR byte = 256-(1000000/sample_rate)
Length of silence	-- in units of sampling cycle
Compression Type	-- of voice data
	8-bits = 0
	4-bits = 1
	2.6-bits = 2
	2-bits = 3
	Multi DAC = 3+(# of channels) [interesting-- this isn’t in the developer’s manual]

RIFF WAVE (.WAV) file format

RIFF is a format by Microsoft and IBM which is similar in spirit and functionality as EA-IFF-85, but not compatible (and it’s in little-endian byte order, of course :-). WAVE is RIFF’s equivalent of AIFF, and its inclusion in Microsoft Windows 3.1 has suddenly made it important to know about.

Rob Ryan was kind enough to send me a description of the RIFF format. Unfortunately, it is too big to include here (27 k), but I’ve made it available for anonymous ftp as ftp.cwi.nl:/pub/audio/RIFF-format.

And here’s a pointer to the official description from Matt Saettler, Microsoft Multimedia:

"The complete definition of the WAVE file format as defined by IBM/Microsoft is available for anon. FTP from ftp.uu.net in the vendor/microsoft/multimedia directory."

(Rob Ryan’s version may actually be an extract from one of the files stored there.)

U-LAW and A-LAW definitions

[Adapted from information provided by duggan@cc.gatech.edu (Rick Duggan) and davep@zenobia.phys.unsw.EDU.AU (David Perry)]

u-LAW (really mu-LAW) is

```
          sgn(m)   (      |m |)       |m |
   y=    ------- ln( 1+ u|--|)       |--| =< 1
         ln(1+u)   (      |mp|)       |mp|
```

A-LAW is

```
     |     A    (m )                 |m |    1
     |  ------- (--)                 |--| =< -
     |  1+ln A  (mp)                 |mp|    A
   y=|
     | sgn(m) (        |m |)    1    |m |
     | ------ ( 1+ ln A|--|)    - =< |--| =< 1
     | 1+ln A (        |mp|)    A    |mp|
```

Values of u=100 and 255, A=87.6, mp is the Peak message value, m is the current quantised message value. (The formulae get simpler if you substitute x for m/mp and sgn(x) for sgn(m); then -1 <= x <= 1.)

Converting from u-LAW to A-LAW is in a sense "lossy" since there are quantizing errors introduced in the conversion.

"..the u-LAW used in North America and Japan, and the
A-LAW used in Europe and the rest of the world and
international routes.."

~References:

Modern Digital and Analog Communication Systems, B.P.Lathi., 2nd ed.
ISBN 0-03-027933-X

Transmission Systems for Communications
Fifth Edition
by Members of the Technical Staff at Bell Telephone Laboratories
Bell Telephone Laboratories, Incorporated
Copyright 1959, 1964, 1970, 1982

A note on the resolution of U-LAW by Frank Klemm <pfk@rz.uni-jena.de>:

8 bit U-LAW has the same lowest magnitude like 12 bit linear and 12 bit U-LAW like 16 linear.

Device/Coding	Resolution	Resolution	
	on maximal level	on low level	
8 bit linear	8	8	
8 bit ulaw	6	12	(used for digital telephone)
12 bit linear	12	12	
12 bit ulaw	10	16	(used in DAT/Longplay)
16 bit linear	16	16	

```
estimated for some analoge technique:
tape recorder (HiFi DIN)
                8                       9      (no Problem today)
tape recorder (semiprofessional)
               10.5                    13.5
```

AVR File Format

~From: hyc@hanauma.Jpl.Nasa.Gov (Howard Chu)

A lot of PD software exists to play Mac .snd files on the ST. One other format that seems pretty popular (used by a number of commercial packages) is the AVR format (from Audio Visual Research). This format has a 128 byte header that looks like this:

```
        char magic[4]="2BIT";
        char name[8];           /* null-padded sample name */
        short mono;             /* 0 = mono, 0xffff = stereo */
        short rez;              /* 8 = 8 bit, 16 = 16 bit */
        short sign;             /* 0 = unsigned, 0xffff = signed */
        short loop;             /* 0 = no loop, 0xffff = looping sample */
        short midi;             /* 0xffff = no MIDI note assigned,
                                   0xffXX = single key note assignment
                                   0xLLHH = key split, low/hi note */
        long rate;              /* sample frequency in hertz */
        long size;              /* sample length in bytes or words (see rez) */
        long lbeg;              /* offset to start of loop in bytes or words.
                                   set to zero if unused. */
        long lend;              /* offset to end of loop in bytes or words.
                                   set to sample length if unused. */
        short res1;             /* Reserved, MIDI keyboard split */
        short res2;             /* Reserved, sample compression */
        short res3;             /* Reserved */
        char ext[20];           /* Additional filename space, used
                                   if (name[7] != 0) */
        char user[64];          /* User defined. Typically ASCII message. */
```

The Amiga MOD Format

~From: norlin@mailhost.ecn.uoknor.edu (Norman Lin)

MOD files are music files containing 2 parts:

(1) a bank of digitized samples
(2) sequencing information describing how and when to play the samples

MOD files originated on the Amiga, but because of their flexibility and the extremely large number of MOD files available, MOD players are now available for a variety of machines (IBM PC, Mac, Sparc Station, etc.)

The samples in a MOD file are raw, 8 bit, signed, headerless, linear digital data. There may be up to 31 distinct samples in a MOD file, each with a length of up to 128K (though most are much smaller; say, 10K - 60K). An older MOD format only allowed for up to 15 samples in a MOD file; you don’t see many of these anymore. There is no standard sampling rate for these samples. [But see below.]

The sequencing information in a MOD file contains 4 tracks of information describing which, when, for how long, and at what frequency samples should be played. This means that a MOD file can have up to 31 distinct (digitized) instrument sounds, with up to 4 playing simultaneously at any given point. This allows a wide variety of orchestrational possibilities, including use of voice samples or creation of one’s own instruments (with appropriate sampling hardware/software). The ability to use one’s own samples as instruments is a flexibility that other music files/formats do not share, and is one of the reasons MOD files are so popular, numerous, and diverse.

15 instrument MODs, as noted above, are somewhat older than 31 instrument MODs and are not (at least not by me) seen very often anymore. Their format is identical to that of 31 instrument MODs except:

(1) Since there are only 15 samples, the information for the last (15th) sample starts at byte 440 and goes through byte 469.
(2) The songlength is at byte 470 (contrast with byte 950 in 31 instrument MOD)
(3) Byte 471 appears to be ignored, but has been observed to be 127. (Sorry, this is from observation only)
(4) Byte 472 begins the pattern sequence table (contrast with byte 952 in a 31 instrument MOD)
(5) Patterns start at byte 600 (contrast with byte 1084 in 31 instrument MOD)

"ProTracker," an Amiga MOD file creator/editor, is available for ftp everywhere as pt??.lzh.

~From: Apollo Wong <apollo@ee.ualberta.ca>

~From: M.J.H.Cox@bradford.ac.uk (Mark Cox)
~Newsgroups: alt.sb.programmer
~Subject: Re: Format for MOD files...
Message-ID: <1992Mar18.103608.4061@bradford.ac.uk>
~Date: 18 Mar 92 10:36:08 GMT
Organization: University of Bradford, UK

wdc50@DUTS.ccc.amdahl.com (Winthrop D Chan) writes:
>I’d like to know if anyone has a reference document on the format of the
>Amiga Sound/NoiseTracker (MOD) files. The author of Modplay said he was going
>to release such a document sometime last year, but he never did. If anyone

I found this one, which covers it better than I can explain it - if you use this in conjunction with the documentation that comes with Norman Lin’s Modedit program it should pretty much cover it.

Mark J Cox

```
/************************************************************************

Protracker 1.1B Song/Module Format:
-----------------------------------

Offset  Bytes  Description
------  -----  -----------
   0     20    Songname. Remember to put trailing null bytes at the end...

Information for sample 1-31:

Offset  Bytes  Description
```

20	22	Samplename for sample 1. Pad with null bytes.
42	2	Samplelength for sample 1. Stored as number of words. Multiply by two to get real sample length in bytes.
44	1	Lower four bits are the finetune value, stored as a signed four bit number. The upper four bits are not used, and should be set to zero.

Value:	Finetune:
0	0
1	+1
2	+2
3	+3
4	+4
5	+5
6	+6
7	+7
8	-8
9	-7
A	-6
B	-5
C	-4
D	-3
E	-2
F	-1

45	1	Volume for sample 1. Range is $00-$40, or 0-64 decimal.
46	2	Repeat point for sample 1. Stored as number of words offset from start of sample. Multiply by two to get offset in bytes.
48	2	Repeat Length for sample 1. Stored as number of words in loop. Multiply by two to get replen in bytes.

Information for the next 30 samples starts here. It’s just like the info for sample 1.

Offset	Bytes	Description
50	30	Sample 2...
80	30	Sample 3...
.		
.		
.		
890	30	Sample 30...
920	30	Sample 31...

Offset	Bytes	Description
950	1	Songlength. Range is 1-128.
951	1	Well... this little byte here is set to 127, so that old trackers will search through all patterns when loading. Noisetracker uses this byte for restart, but we don’t.
952	128	Song positions 0-127. Each hold a number from 0-63 that tells the tracker what pattern to play at that position.
1080	4	The four letters "M.K." - This is something Mahoney & Kaktus inserted when they increased the number of samples from 15 to 31. If it’s not there, the module/song uses 15 samples or the text has been removed to make the module harder to rip. Startrekker puts "FLT4" or "FLT8" there instead.

Offset	Bytes	Description
1084	1024	Data for pattern 00.

```
   .
   .
   .
xxxx  Number of patterns stored is equal to the highest patternnumber
      in the song position table (at offset 952-1079).
```

Each note is stored as 4 bytes, and all four notes at each position in the pattern are stored after each other.

```
00 -  chan1  chan2  chan3  chan4
01 -  chan1  chan2  chan3  chan4
02 -  chan1  chan2  chan3  chan4
etc.
```

Info for each note:

```
 _____byte 1_____   byte2_    _____byte 3_____   byte4_
/                \ /      \   /                \ /      \
0000          0000-00000000  0000          0000-00000000

Upper four    12 bits for    Lower four    Effect command.
bits of sam-  note period.   bits of sam-
ple number.                  ple number.
```

```
Periodtable for Tuning 0, Normal
  C-1 to B-1 : 856,808,762,720,678,640,604,570,538,508,480,453
  C-2 to B-2 : 428,404,381,360,339,320,302,285,269,254,240,226
  C-3 to B-3 : 214,202,190,180,170,160,151,143,135,127,120,113
```

To determine what note to show, scan through the table until you find the same period as the one stored in byte 1-2. Use the index to look up in a notenames table.

This is the data stored in a normal song. A packed song starts with the four letters "PACK", but i don't know how the song is packed: You can get the source code for the cruncher/decruncher from us if you need it, but I don't understand it; I've just ripped it from another tracker...

In a module, all the samples are stored right after the patterndata. To determine where a sample starts and stops, you use the sampleinfo structures in the beginning of the file (from offset 20). Take a look at the mt_init routine in the playroutine, and you'll see just how it is done.

Lars "ZAP" Hamre/Amiga Freelancers

```
***********************************************************************/
```

--
Mark J Cox -----
Bradford, UK ---

PS: A file with even *much* more info on MOD files, compiled by Lars Hamre, is available from ftp.cwi.nl:/pub/audio/MOD-info. Enjoy!

FTP sites for MODs and MOD players

~Subject: MODS AND PLAYERS!! **READ** info/where to get them

```
~From: cjohnson@tartarus.uwa.edu.au (Christopher Johnson)
~Newsgroups: alt.binaries.sounds.d
Message-ID: <1h32ivINNglu@uniwa.uwa.edu.au>
~Date: 21 Dec 92 00:19:43 GMT
Organization: The University of Western Australia

Hello world,

For all those asking, here is where to get those mod players and mods.

SNAKE.MCS.KENT.EDU is the best site for general stuff.  look in /pub/SB-Adlib

Simtel-20 or archie.au(simtel mirror) in <msdos.sound>

for windows players ftp.cica.indiana.edu in pub/pc/win3/sound

here is a short list of players

mp or modplay   BEST OVERALL                    mp219b.zip
        simtel and snake

wowii           best for vga/fast machines      wowii12b.zip
        simtel and snake

trakblaster     best for compatability          trak-something
        simtel and snake        two versions, old one for slow
        machines

ss              cute display(hifi)              have_sex.arj
        found on local BBS (western Australia White Ghost)

superpro player generally good                  ssp.zip or similar
        found on night owl 7 CD

player?         cute display(hifi)              player.zip or similar
        found on night owl 7 CD

WINDOWS

Winmod pro      does protracker                 wmp????.zip
        cica

winmod          more stable                     winmod12.zip or similar
        cica

Hope this helps, e-mail me if you find any more players and I will add them in for t
little out of hand.

for mods ftp to wuarchive.wustl.edu and go to the amiga music directory (pub/amiga/m
a while

see you soon

Chris.

-----------------------------------------------------------------------
```

Site Map

Minilab

Perimeter Detection

Cloud to CAD

Visual Object Recognition

Rapid World Modeling

Standoff Sensor

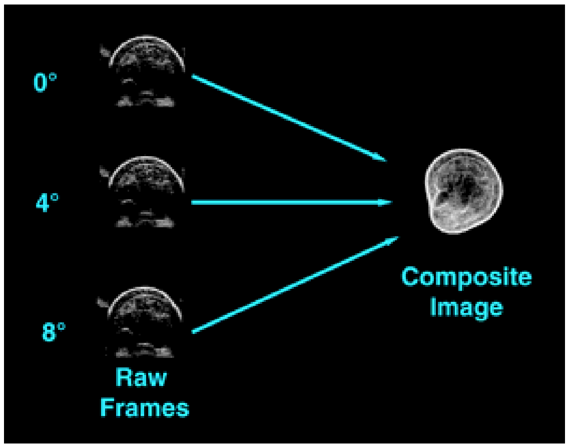
TMSS

3D Ultrasound Imaging

High Definition 3D Ultrasound Imaging

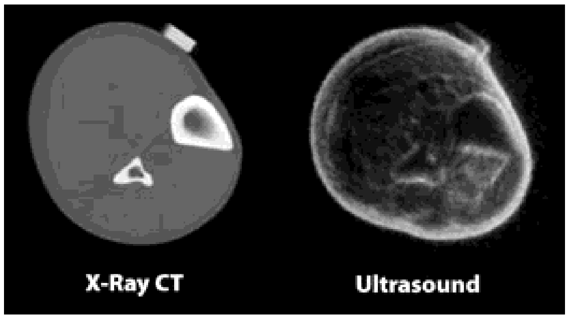
Description

The 3D Ultrasound Imaging System developed by Sandia National Laboratories combines robotic controls, image processing, and artificial intelligence to provide the basis for improved resolution for medical diagnostic imaging applications. Our system takes a series of raw ultrasound scans and combines them to form a 2D slice. Slices are stacked to form a 3D volume dataset. The 3D anatomical geometry has been used in the past for custom fabrication of prosthetic limbs for amputees.



Forming composite ultrasound image from conventional ultrasound

Ultrasound is highly specular: it depends on angle of incidence for contrast and resolution. By scanning around the target we change the angle of incidence and obtain redundant imaging of the same areas. The images are much improved over conventional ultrasound by using a patent-pending technique of optimizing the signal uniformly throughout the image plane. Image processing also removes patient motion artifacts. Techniques for image processing are derived from technologies used in satellite imaging and radar for the military.



CT versus composite ultrasound image

3D ultrasound imaging has particular value for information gathering. Ultrasound equipment can be one-tenth the cost of computed tomography (CT) or magnetic resonance imaging (MRI) machines and does not require specially built facilities for radiation protection. In addition, ultrasound is a non-ionizing, safer form of radiation compared with x-ray CT.

Where we'd like to go

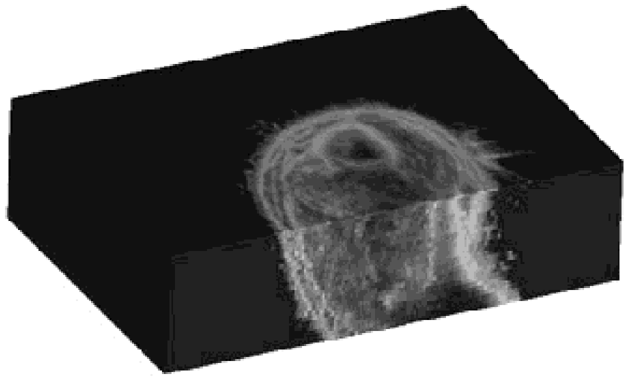
It is our goal to demonstrate that our methods of ultrasound imaging can be used to diagnose and possibly detect breast tumors at the same stage as is currently possible with mammography, but without the pain of compression or the perceived harmful effects of ionizing radiation. In addition, the methods can be used to form 3D images of the breast for complete mapping of breast tissue and tumors. 3D imaging will provide details of breast cancers that can be analyzed and compared from year to year with much higher confidence. Improvements seen in 3D ultrasound over conventional ultrasound will be similar to those gained by x-ray CT over flat plate x-ray.

This effort is a collaboration with university researchers, military researchers, and industry. Medical researchers, including physicians and scientists, have praised the work as leading edge research that moves ultrasound to the next level of value in diagnostics. Military researchers value the technology for applications in combat casualty care. Industry leaders see an opportunity to improve their imaging products and market competitiveness.

Sandia is currently collaborating with the University of New Mexico Health Sciences Center and various ultrasound industry leaders.

Features

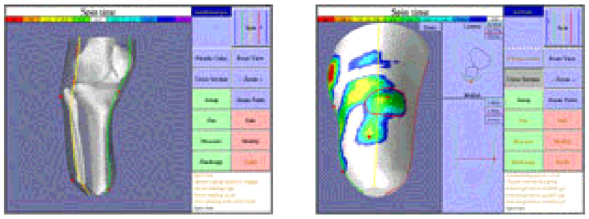
- 3D ultrasound images
- Lower cost, greater portability, higher resolution than conventional imaging



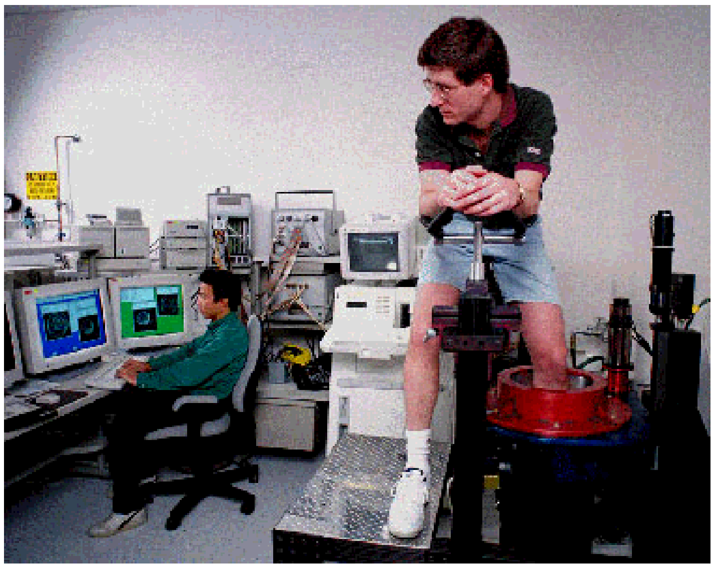
Applications

- Prosthesis fabrication

- Manufacturing, nondestructive evaluation, and testing
- Antiterrorist mine and bomb imaging
- Military combat casualty care diagnostic imaging
- Primary and tertiary care medical facilities



Prosthesis fabrication



Contact:
Charles Q. Little
(505) 284-3151
email: cqlittl@sandia.gov

Last modified: 11/2/99 G. Garrison

Back to top

Search

[Home] [Capabilities] [Resources] [Working with Us]

```
Archive-name: audio-fmts/part1
Submitted-by: Guido van Rossum
Version: 3.04
Last-modified: 19-Jul-1993
```

FAQ: Audio File Formats

```
Table of contents
-----------------
```

Introduction

This is version 3 of this FAQ, which I started in November 1991 under the name "The audio formats guide". I bumped the major version number again at the occasion of the split in two parts: part one is the main text and part two consists of the collection of appendices.

I am posting this about once a fortnight, either unchanged (just to inform new readers), or updated (if I learn more or when new hardware or software becomes popular). I post to alt.binaries.sounds.{misc,d} and to comp.dsp, for maximal coverage of people interested in audio, and to {news,comp}.answers, for easy reference.

The entire FAQ is also available by anonymous ftp from ftp.cwi.nl [192.16.184.180], directory pub/audio, files AudioFormats.{part1,part2}.

BTW: All FAQs, including this one, are available for anonymous ftp on the archive site rtfm.mit.edu in directory /pub/usenet/news.answers/. The name under which a FAQ is archived appears in the "Archive-Name:" line at the top of the article. This FAQ is archived as audio-fmts/part1.

A companion posting with subject "Changes to: ..." is occasionally posted listing the diffs between a new version and the last. This is not reposted, and it is suppressed when the diffs are bigger than the new version.

This FAQ is also available in distributed hypertext form. If you have a WWW browser and direct Internet access you can point it to "http://voorn.cwi.nl/audio-formats/a00.html". (WWW is the CERN World-Wide Web initiative; for more info, telnet or ftp to info.cern.ch.) You can also ftp the files from ftp.cwi.nl, directory /pub/www/audio-formats/. (Due to lack of time for maintenance, the WWW version is currently out of date. Sorry.)

Send updates, comments and questions to . I'd like to thank everyone who sent updates in the past.

--Guido van Rossum, CWI, Amsterdam

```
Device characteristics
----------------------

In this text, I will only use the term "sample" to refer to a single
output value from an A/D converter, i.e., a small integer number
(usually 8 or 16 bits).

Audio data is characterized by the following parameters, which
correspond to settings of the A/D converter when the data was
recorded.  Naturally, the same settings must be used to play the data.

- sampling rate (in samples per second), e.g. 8000 or 44100

- number of bits per sample, e.g. 8 or 16

- number of channels (1 for mono, 2 for stereo, etc.)

Approximate sampling rates are often quoted in Hz or kHz ([kilo-]
Hertz), however, the politically correct term is samples per second
(samples/sec).  Sampling rates are always measured per channel, so for
stereo data recorded at 8000 samples/sec, there are actually 16000
samples in a second.  I will sometimes write 8 k as a shorthand for
8000 samples/sec.

Multi-channel samples are generally interleaved on a frame-by-frame
basis: if there are N channels, the data is a sequence of frames,
where each frame contains N samples, one from each channel.  (Thus,
the sampling rate is really the number of *frames* per second.)  For
stereo, the left channel usually comes first.

The specification of the number of bits for U-LAW (pronounced mu-law
-- the u really stands for the Greek letter mu) samples is somewhat
problematic.  These samples are logarithmically encoded in 8 bits,
like a tiny floating point number; however, their dynamic range is
that of 12 bit linear data.  Source for converting to/from U-LAW
(written by Jef Poskanzer) is distributed as part of the SOX package
mentioned below; it can easily be ripped apart to serve in other
applications.  The official definition is the CCITT standard G.711.

There exists another encoding similar to U-LAW, called A-LAW, which
is used as a European telephony standard.  There is less support for
it in UNIX workstations.
```

(See the Appendix for some formulae describing U-LAW and A-LAW.)

Popular sampling rates

Some sampling rates are more popular than others, for various reasons. Some recording hardware is restricted to (approximations of) some of these rates, some playback hardware has direct support for some. The popularity of divisors of common rates can be explained by the simplicity of clock frequency dividing circuits :-).

Samples/sec	Description
5500	One fourth of the Mac sampling rate (rarely seen).
7333	One third of the Mac sampling rate (rarely seen).
8000	Exactly 8000 samples/sec is a telephony standard that goes together with U-LAW (and also A-LAW) encoding. Some systems use an slightly different rate; in particular, the NeXT workstation uses 8012.8210513, apparently the rate used by Telco CODECs.
11 k	Either 11025, a quarter of the CD sampling rate, or half the Mac sampling rate (perhaps the most popular rate on the Mac).
16000	Used by, e.g. the G.722 compression standard.
18.9 k	CD-ROM/XA standard.
22 k	Either 22050, half the CD sampling rate, or the Mac rate; the latter is precisely 22254.545454545454 but usually misquoted as 22000. (Historical note: 22254.5454... was the horizontal scan rate of the original 128k Mac.)
32000	Used in digital radio, NICAM (Nearly-Instantaneous Companded Audio Multiplex [IBA/BREMA/BBC]) and other TV work, at least in the UK; also long play DAT and Japanese HDTV.
37.8 k	CD-ROM/XA standard for higher quality.
44056	This weird rate is used by professional audio equipment to fit an integral number of samples in a video frame.
44100	The CD sampling rate. (DAT players recording digitally from CD also use this rate.)
48000	The DAT (Digital Audio Tape) sampling rate for domestic use.

Files samples on SoundBlaster hardware have sampling rates that are divisors of 1000000.

While professinal musicians disagree, most people don’t have a problem if recorded sound is played at a slightly different rate, say, 1-2%. On the other hand, if recorded data is being fed into a playback device in real time (say, over a network), even the smallest difference in sampling rate can frustrate the buffering scheme used...

There may be an emerging tendency to standardize on only a few sampling rates and encoding styles, even if the file formats may differ. The suggested rates and styles are:

```
    rate (samp/sec) style mono/stereo

    8000 8-bit U-LAW mono
    22050 8-bit linear unsigned mono and stereo
    44100 16-bit linear signed mono and stereo
```

Compression schemes

Strange though it seems, audio data is remarkably hard to compress effectively. For 8-bit data, a Huffman encoding of the deltas between successive samples is relatively successful. For 16-bit data, companies like Sony and Philips have spent millions to develop proprietary schemes. Information about PASC (Philips’ scheme) can be found in Advanced Digital Audio by Ken C. Pohlmann.

Public standards for voice compression are slowly gaining popularity, e.g. CCITT G.721 (ADPCM at 32 kbits/sec) and G.723 (ADPCM at 24 and 40 kbits/sec). (ADPCM == Adaptive Delta Pulse Code Modulation.) Sun Microsoft has placed the source code of a portable implementation of these algorithms (as well as G.711, which defines A-LAW and U-LAW) in the public domain (needless to say, their proprietary implementation distributed in binary form with Solaris is better :-). One place to ftp this source code from is ftp.cwi.nl:/pub/audio/ccitt-adpcm.tar.Z. Source for another 32 kbits/sec ADPCM implementation, assumed to be compatible with Intel’s DVI audio format, can be ftp’ed from ftp.cwi.nl:/pub/audio/adpcm.shar. (** NOTE: if you are using v1.0, you should get v1.1, released 17-Dec-1992, which fixes a serious bug -- the quality of v1.1 is claimed to be better than U-LAW **)

GSM 06.10 is a speech encoding in use in Europe that compresses 160 13-bit samples into 260 bits (or 33 bytes), i.e. 1650 bytes/sec (at 8000 samples/sec). A free implementation can be ftp’ed from tub.cs.tu-berlin.de, file /pub/tubmik/gsm-1.0.tar.Z.

There are also two US federal standards, 1016 (Code excited linear prediction (CELP), 4800 bits/s) and 1015 (LPC-10E, 2400 bits/s). See also the appendix for 1016.

Tony Robinson has written a good FAST loss-less compression for lots of different audio formats (particularly good for WAV and MOD files). The software is available by anonymous ftp from svr-ftp.eng.cam.ac.uk [129.169.24.20], directory misc, file shorten-1.08.tar.Z.

(Note that U-LAW and silence detection can also be considered compression schemes.)

Here’s a note about audio codings by Van Jacobson :
Several people used the words "LPC" and "CELP" interchangably. They are very different. An LPC (Linear Predictive Coding) coder fits speech to a simple, analytic model of the vocal tract, then throws away the speech & ships the parameters of the best-fit model. An LPC

decoder uses those parameters to generate synthetic speech that is usually more-or-less similar to the original. The result is intelligible but sounds like a machine is talking. A CELP (Code Excited Linear Predictor) coder does the same LPC modeling but then computes the errors between the original speech & the synthetic model and transmits both model parameters and a very compressed representation of the errors (the compressed representation is an index into a ’code book’ shared between coders & decoders -- this is why it’s called "Code Excited"). A CELP coder does much more work than an LPC coder (usually about an order of magnitude more) but the result is much higher quality speech: The FIPS-1016 CELP we’re working on is essentially the same quality as the 32Kb/s ADPCM coder but uses only 4.8Kb/s (the same as the LPC coder).

The comp.compression FAQ has some text on the 6:1 audio compression scheme used by MPEG (a video compression standard-to-be). It’s interesting to note that video compression reaches much higher ratios (like 26:1). This FAQ is ftp’able from rtfm.mit.edu [18.72.1.58] in directory /pub/usenet/news.answers/compression-faq, files part1 and part2.

Comp.compression also carries a regular posting "How to uncompress anything" by David Lemson , which (tersely) hints on which program you need to uncompress a file whose name ends in . for almost any conceivable . Ftp’able from ftp.cso.uiuc.edu (128.174.5.59) in the directory /doc/pcnet as the file compression.

Documentation on a digital cellular telephone system by Qualcomm Inc. can be ftp’ed from ftp.qualcomm.com:/pub/cdma; the vocoder is in appendix A.

Apple has an Audio Compression/Expansion scheme called ACE (on the GS) / MACE (on the Macintosh). It’s a lossy scheme that attempts to predict where the wave will go on the next sample. There’s very little quality change on 8:4 compression, somewhat more for 8:3. It does guarantee exactly 50% or 62.5% compression, though. I believe MACE uses larger ratios/more loss, but I’m unsure of the specific numbers. (Marc Sira)

Current hardware

I am aware of the following computer systems that can play back and (sometimes) record audio data, with their characteristics. Note that for most systems you can also buy "professional" sampling hardware, which supports much better quality, e.g. >= 44.1 k 16 bits stereo. The characteristics listed here are a rough estimate of the capabilities of the basic hardware only (and even here I am on thin ice, with systems becoming ever more powerful).

machine	bits	max sampling rate	#output channels
Mac (all types)	8	22k	1
Mac (newer ones)	16	64k	4(128)
Apple IIgs	8	32k / >70k	16(st)
PC/Soundblaster v1	8	13k / 22k	1
PC/Soundblaster v2	8	15k / 44.1k	1
PC/PAS-16	16	44.1k	?(st)
Atari ST	8	22k	1
Atari STe,TT	8	50k	2

Atari Falcon 030	16	50k	8(st)
Amiga	8	varies above 29k	4(st)
Sun Sparc	U-LAW	8k	1
Sun Sparcst. 10	U-LAW,8,16	48k	1(st)
NeXT	U-LAW,8,16	44.1k	1(st)
SGI Indigo	8,16	48k	4(st)
SGI Indigo2	8,16	48k	16(st,4-channel)
Acorn Archimedes	~U-LAW	~180k	8(st)
Sony NWS-3xxx	U,A,8,16	8-37.8k	1(st)
Sony NWS-5xxx	U,A,8,16	8-48k	1(st)
VAXstation 4000	U-LAW	8k	1
DEC 3000/300-500	U-LAW	8k	1
DEC 5000/20-25	U-LAW	8k	1
Tandy 1000/*L*	8	22k	3
Tandy 2500	8	22k	3
HP9000/705,710,425e	U,A-LAW,16	8k	1
HP9000/715,725,735	U,A-LAW,16	48k	1(st)
HP9000/755 option:	U,A-LAW,16	48k	1(st)

4(st) means "four voices, stereo"; sampling rates xx/yy are different recording/playback rates; *L* is any type with 'L' in it.

All these machines can play back sound without additional hardware, although the needed software is not always standard; also, some machines need external hardware to record sound (or to record at higher quality, like the NeXT, whose built-in sampling hardware only does 8000 samples/sec in U-LAW). Please don't send me details on optional or 3rd party hardware, there is too much and it is really beyond the scope of this FAQ. In particular, there is a separate newsgroup devoted to PC sound cards: comp.sys.ibm.pc.soundcard, which includes FAQ of its own.

The new VAXstation 4000 (VLC and model 60) series lets you PLAY audio (.au) files, and the package DECsound will let you do the recording. In fact, DECsound is given away free with Motif 1.1 and supports the VAXstation, Sun SPARCstation, DECvoice, and DECaudio devices. Sun sound files work without change. The Alpha systems (DEC 3000 Model 300, 400, 500) also have DECsound bundled with Motif.

Notes for the DECstation 5000/20-25: You need either XMedia tools from DEC ($$$$), or the AudioFile package (which works nicely) from crl.dec.com (see below). The audio device is "/dev/bba", you cannot send ".au" files directly to the device, the Xmedia/AF software provide an "audioserver" which must be run to play/record sounds.

The SGI Personal IRIS 4D/30 and 4D/35 have the same capabilities as the Indigo. The audio board was optional on the 4D/30.
The Indigo2 features are a superset of the Indigo features.

The new Apple Macs have more powerful audio hardware; the latest models have built-in microphones.

Software exists for the PC that can play sound on its 1-bit speaker using pulse width modulation (see appendix); the Soundblaster board records at rates up to 13 k and plays back up to 22 k (weird combination, but that's the way it is).

Here's some info about the newest Atari machine, the Falcon030. This machine has stereo 16 bit CODECs and a 32 MHz Motorola 56001 that can handle 8 channels of 16 bit audio, up to 50 khz/channel with simultaneous playback and record. The Falcon DMA sound engine is also compatible with the 8 bit stereo DMA used on the STe and TT. All of these systems use signed data.

On the NeXT, the Motorola 56001 DSP chip is programmable and you can (in principle) do what you want. The SGI Indigo uses the same DSP chip but it can't be programmed by users -- SGI prefers to offer it as a shared system resource to multiple applications, thus enabling developers to program audio with their Audio Library and avoid code modifications for execution on future machines with different audio hardware, i.e. a different DSP. For example, the Indigo2 does not have a DSP chip.

The Amiga also has a 6-bit volume, which can be used to produce something like a 14-bit output for each voice. The hardware can also use one of each voice-pair to modulate the other in FM (period) or AM (volume, 6-bits).

The Acorn Archimedes uses a variation on U-LAW with the bit order reversed and the sign bit in bit 0. Being a 'minority' architecture, Arc owners are quite adept at converting sound/image formats from other machines, and it is unlikely that you'll ever encounter sound in one of the Arc's own formats (there are several).

CD-I machines form a special category. The following formats are used:

- PCM 44.1 kHz standard CD format
- ADPCM - Addaptive Delta PCM
  - Level A 37.8 kHz 8-bit
  - Level B 37.8 kHz 4-bit
  - Level C 18.9 kHz 4-bit

File formats

Historically, almost every type of machine used its own file format for audio data, but some file formats are more generally applicable, and in general it is possible to define conversions between almost any pair of file formats -- sometimes losing information, however.

File formats are a separate issue from device characteristics. There are two types of file formats: self-describing formats, where the device parameters and encoding are made explicit in some form of header, and "raw" formats, where the device parameters and encoding are fixed.

Self-describing file formats generally define a family of data encodings, where a header fields indicates the particular encoding variant used. Headerless formats define a single encoding and usually allows no variation in device parameters (except sometimes sampling rate, which can be a pain to figure out other than by listening to the sample).

The header of self-describing formats contains the parameters of the sampling device and sometimes other information (e.g. a human-readable description of the sound, or a copyright notice). Most headers begin with a simple "magic word". (Some formats do not simply define a header format, but may contain chunks of data intermingled with chunks of encoding info.) The data encoding defines how the actual samples are stored in the file, e.g. signed or unsigned, as bytes or short integers, in little-endian or big-endian byte order, etc. Strictly spoken, channel interleaving is also part of the encoding, although so far I have seen little variation in this area.

Some file formats apply some kind of compression to the data, e.g. Huffman encoding, or simple silence deletion.

Here’s an overview of popular file formats.

Self-describing file formats

extension, name	origin	variable parameters (fixed; comments)
.au or .snd	NeXT, Sun	rate, #channels, encoding, info string
.aif(f), AIFF	Apple, SGI	rate, #channels, sample width, lots of info
.aif(f), AIFC	Apple, SGI	same (extension of AIFF with compression)
.iff, IFF/8SVX	Amiga	rate, #channels, instrument info (8 bits)
.voc	Soundblaster	rate (8 bits/1 ch; can use silence deletion)
.wav, WAVE	Microsoft	rate, #channels, sample width, lots of info
.sf	IRCAM	rate, #channels, encoding, info
none, HCOM	Mac	rate (8 bits/1 ch; uses Huffman compression)
none, MIME	Internet	(see below)
.mod or .nst	Amiga	(see below)

Note that the filename extension ".snd" is ambiguous: it can be either the self-describing NeXT format or the headerless Mac/PC format, or even a headerless Amiga format.

I know nothing for sure about the origin of HCOM files, only that there are a lot of them floating around on our system and probably at FTP sites over the world. The filenames usually don’t have a ".hcom" extension, but this is what SOX (see below) uses. The file format recognized by SOX includes a MacBinary header, where the file type field is "FSSD". The data fork begins with the magic word "HCOM" and contains Huffman compressed data; after decompression it it is 8 bits unsigned data.

IFF/8SVX allows for amplitude contours for sounds (attack/decay/etc). Compression is optional (and extensible); volume is variable; author, notes and copyright properties; etc.

AIFF, AIFC and WAVE are similar in spirit but allow more freedom in encoding style (other than 8 bit/sample), amongst others.

There are other sound formats in use on Amiga by digitizers and music programs, such as IFF/SMUS.

Appendices describes the NeXT and VOC formats; pointers to more info about AIFF, AIFC, 8SVX and WAVE (which are too complex to describe here) are also in appendices.

DEC systems (e.g. DECstation 5000) use a variant of the NeXT format that uses little-endian encoding and has a different magic number (0x0064732E in little-endian encoding).

Standard file formats used in the CD-I world are IFF but on the disc they’re in realtime files.

An interesting "interchange format" for audio data is described in the proposed Internet Standard "MIME", which describes a family of transport encodings and structuring devices for electronic mail. This is an extensible format, and initially standardizes a type of audio data dubbed "audio/basic", which is 8-bit U-LAW data sampled at 8000 samples/sec.

The "IRCAM" sound file system has now been superseded by the so-called "BICSF" (for Berkeley/IRCAM/CARL Sound File system) software release. More recently, there has been an effort at Princeton (Prof. Paul Lansky) and Stanford (Stephen Travis Pope) to standardize several extensions to BICSF. A description of BICSF and the Princeton/Stanford extensions is available by anonymous ftp from ftp.cwi.nl [192.16.184.180], in directory /pub/audio/BICSF-info. This file contains further ftp pointers to software.

Finally, a somewhat different but popular format are "MOD" files, usually with extension ".mod" or ".nst" (they can also have a prefix of "mod."). This originated at the Amiga but players now exist for many platforms. MOD files are music files containing 2 parts: (1) a bank of digitized samples; (2) sequencing information describing how and when to play the samples. See the appendix "The Amiga MOD Format" for a description of this file format (and pointers to ftp'able players and example MOD files).

Headerless file formats

extension or name	origin	parameters
.snd, .fssd	Mac, PC	variable rate, 1 channel, 8 bits unsigned
.ul	US telephony	8 k, 1 channel, 8 bit "U-LAW" encoding
.snd?	Amiga	variable rate, 1 channel, 8 bits signed

It is usually easy to distinguish 8-bit signed formats from unsigned by looking at the beginning of the data with 'od -b)

Sound Format	SOX/DOS file ext	MAC type	Mac program: convert to 'snd'
Mac snd	.snd	sfil	n/a
Amiga IFF/8SVX	.iff		AmigaSndConverter
Amiga SoundTracker	.mod	STrk	ModVoicer
Audio IFF	.aiff	AIFF	SoundExtractor, Sample Editor, UUTool
DSP Designer		DSPs	SoundHack
IRCAM	.sf	IRCM	SoundHack
MacMix		MSND	SoundHack
RIFF WAVE	.wav		SoundExtractor
SoundBlaster	.voc		SoundExtractor
SoundDesigner/AudioMedia		Sd2f	SoundHack
Sound[Edit|Cap|Wave]	.hcom	FSSD	SoundExtractor, SoundEdit Wavicle
Sun uLaw/Next .snd	.au/.snd	NxTS	SoundExtractor, SoundHack au<->snd, UUTool

File conversions

SOX (UNIX, PC, Amiga)

The most versatile tool for converting between various audio formats is SOX ("Sound Exchange"). It can read and write various types of audio files, and optionally applies some special effects (e.g. echo, channel averaging, or rate conversion).

SOX recognizes all filename extensions listed above except ".snd", which would be ambiguous anyway, and ".wav" (but there’s a patch, see below). Use type ".au" for NeXT ".snd" files. Mac and PC ".snd" files are completely described by these parameters:

```
        -t raw -b -u -r 11000
```

(or -r 22000 or -r 7333 or -r 5500; 11000 seems to be the most common rate).

The source for SOX, version 6, platchlevel 8, was posted to alt.sources, and should be widely archived. (Patch 9 was posted later and incporporates some important .wav fixes.) To save you the trouble of hunting it down, it can be gotten by anonymous ftp from wuarchive.wustl.edu, in the directory usenet/alt.sources/articles, files 7288.Z through 7295.Z. (These files are compressed news articles containing shar files, if you hadn’t guessed.) I am sure many sites have similar archives, I’m just listing one that I know of and which carries a lot of this kind of stuff. (Also see the appendix if you don’t have Internet access.)

A compressed tar file containing the same version of SOX is available by anonymous ftp from ftp.cwi.nl [192.16.184.180], in directory /pub/audio/sox7.tar.Z. You may be able to locate a nearer version using archie!

Ports of SOX:

- The source as posted should compile on any UNIX and PC system.

- A PC version is available by ftp from ftp.cwi.nl (see above) as pub/audio/sox5dos.zip; also available from the garbo mail server.

- The latest Amiga SOX is available via anonymous ftp to wuarchive.wustl.edu, files systems/amiga/audio/utils/amisox*. (See below for a non-SOX solution.)
  The final release of r6 will compile as distributed on the Amiga with SAS/C version 6. Binaries (since many Amiga users do not own compilers) will continue to be available for FTP.

SOX usage hints:

- Often, the filename extension of sound files posted on the net is wrong. Don’t give up, try a few other possibilities using the "-t " option. Remember that the most common file type is unsigned bytes, which can be indicated with "-t ub". You’ll have to guess the proper sampling rate, but often it’s 11k or 22k.

- In particular, with SOX version 4 (or earlier), you have to specify "-t 8svx" for files with an .iff extension.

- When converting linear samples to U-LAW using the .au type for the output file, you must specify "-U" for the output file, otherwise you will end up with a file containing a NeXT/Sun header but linear samples -- only the NeXT will play such files correctly. Also, you must explicitly specify an output sampling rate with "-r 8000". (This may seem fixed for most cases in version 5, but it is still occasionally necessary, so I'm keeping this warning in.)

Sun Sparc

On Sun Sparcs, starting at SunOS 4.1, a program "raw2audio" is provided by Sun (in /usr/demo/SOUND -- see below) which takes a raw U-LAW file and turns it into a ".au" file by prefixing it with an appropriate header.

NeXT

On NeXTs, you can usually rename .au files to .snd and it'll work like a charm, but some .au files lack header info that the NeXT needs. This can be fixed by using sndconvert:

```
        sndconvert -c 1 -f 1 -s 8012.8210513 -o nextfile.snd sunfile.au
```

SGI Indigo and Personal IRIS

SGI supports "soundfiler" (in /usr/sbin), a program similar in spirit to SOX but with a GUI. Soundfiler plays aiff, aifc, NeXT/Sun and .wav formats. It can do conversions between any of these formats and to and from raw formats including mulaw. It also does sample rate conversions.

Three shell commands are also provided that give the same functionality: "sfplay", "sfconvert", and "aifcresample" (all in /usr/sbin).

Amiga

Mike Cramer's SoundZAP can do no effects except rate change and it only does conversions to IFF, but it is generally much faster than SOX. (Ftp'able from the same directory as amisox above.)

Newer versions of OmniPlay (see below) will also convert to IFF.

Tandy

The Tandy 1000 uses a (proprietary?) compressed format. There is a PD Mac to Tandy conversion program called CONVERT. Leonard Erickson
 writes: There is a WAV driver from Tandy
if people ask. There also appears to be a program that purports to convert other formats to Tandy, but I haven't tested this one yet.

Apple Macintosh

Bill Houle sent the following list:

SoundHack, Tom Erbe: can read/write Sound Designer II, Audio IFF, IRCAM, DSP Designer and NeXT .snd (or Sun .au); 8-bit uLaw, 8-bit linear, 32-bit floating point and 16-bit linear data encoding. Can read (but not write) raw data files. Implements soundfile convolution, a phase vocoder, a binaural filter and an amplitude analysis & gain change module.

AmigaSndConverter, Povl H. Pederson: converts Amiga IFF/8SVX to Mac ’snd’.

SoundExtractor, Alberto Ricci: extracts ’snd’ resources, AIFF, SoundEdit, VOC, and WAV data from practically anything, converting to ’snd’ files.

au<->Mac, Victor J. Heinz: converts Sun uLaw files to Mac ’snd’ files.

UUTool, Bernie Wieser: primarily a uuencode/decode program, but can also read/write Sun uLaw, AIFF, and ’snd’ files.

ModVoicer, Kip Walker: converts Amiga MOD voices into SoundEdit files or ’snd’ resources.

Most programs mentioned are shareware/freeware available from SUMEX and the various mirror sites, or check archie for the nearest FTP site.

Playing audio files on UNIX

The commands needed to play an audio file depend on the file format and the available hardware and software. Most systems can only directly play sound in their native format; use a conversion program (see above) to play other formats.

Sun Sparcstation running SunOS 4.x

Raw U-LAW files can be played using "cat file >/dev/audio".

A whole package for dealing with ".au" files is provided by Sun on an experimental basis, in /usr/demo/SOUND. You may have to compile the programs first. (If you can’t find this directory, either you are not running SunOS 4.1 yet, or your system administrator hasn’t installed it -- go ask him for it, not me!) The program "play" in this directory recognizes all files in Sun/NeXT format, but a SS 1 or 2 can play only those using U-LAW encoding at 8 k -- the SS 10 hardware plays other encodings, too.

If you ca’t find "play", you can also cat a ".au" file to /dev/audio, if it uses U-LAW; the header will sound like a short burst of noise but the rest of the data will sound OK (really, the only difference in this case between raw U-LAW and ".au" files is the header; the U-LAW data is exactly the same).

Finally, OpenWindows 3.0 has a full-fledged audio tool. You can drop audio file icons into it, edit them, etc.

Sun Sparcstation running Solaris 2.0

Under SVR4 (and hence Solaris 2.0), writing to /dev/audio from the shell is a bad idea, because the device driver will flush its queue as soon as the file is closed. Use "audioplay" instead. The supported formats and sampling rates are the same as above.

NeXT

On NeXT machines, the standard "sndplay" program can play all NeXT

format files (this include Sun ".au" files). It supports at least
U-LAW at 8 k and 16 bits samples at 22 or 44.1 k. It attempts
on-the-fly conversions for other formats.

Sound files are also played if you double-click on them in the file
browser.

SGI Indigo and Personal IRIS

On SGI Indigo, Indigo2 and the 4D/30 and /35 Personal IRIS workstations,
"WorkSpace" plays audio files in .aiff, .aifc, .au, and .wav formats if
you double click them and the sampling rate is one of 8000, 11025,
16000, 22050, 32000, 44100, or 48000. On the Personal IRIS, you need
to have the audio board installed (check the output from hinv) and you
must run IRIX 3.3.2 or 4.0 or higher. These files can also be played
with "soundfiler" and "sfplay". ".aiff" and ".aifc" files at the above
sampling rates can also be played with playaifc. (All in /usr/sbin)

There is no simple /dev/audio interface on these SGI machines. (There
was one on 4D/25 machines, reading and writing signed linear 8-bit
samples at rates of 8, 16 and 32 k.)

A program "playulaw" was posted as part of the "radio 2.0" release
that I posted to several source groups; it plays raw U-LAW files on
the Indigo, Indigo2 or Personal IRIS audio hardware.

Sony NEWS

The Sony RISC-NEWS line (NWS-3250 laptop, NWS-37xx desktop, NWS-38xx
desktop w/ IOP) also has builtin sound capabilities. You can also buy
external boards for the older NEWS machines or to add extra channels
to the new machines. In the default mode (8k/8-bit), Sun .au files
are directly supported (you can ’cat’ .au files to /dev/sb and have
them play).

Others

Most other UNIX boxes don’t have audio hardware and thus can’t play
audio data. This is actually rapidly changing and most new hardware
that hits the market has some form of audio support. Unfortunately
there is nothing like X11 that provides a portable interface. Perhaps
DEC CRL’s AudioFile system can fill the gap; it is network-transparent
and supports at least Digital RISC systems running Ultrix, Digital
Alpha AXP systems running OSF/1, Sun Microsystems SPARCstations
running SunOS, and SGI Indigos. The source kit is located at ftp site
crl.dec.com (Internet 192.58.206.2) in /pub/DEC/AF.

Playing audio files on the Vaxstation 4000 (VMS)

1) Without DECsound

".au" files can be played by COPYING them to device "SOA0:". This
device is set up by enabling the driver SODRIVER. You can use the
following command file:

```
$!---------------- cut here -------------------------------
```

```
$! sound_setup.com    enable SOUND driver
$ run sys$system:sysgen
connect soa0 /adapter=0 /csr=%x0e00 /vector=%o304 /driver=sodriver
exit
$ exit
$!----------------- cut here -----------------------------------
```

2) With DECsound (bundled with motif)

Just start DECsound by selecting it from the session manager in the applications menu. (Not there use "@vue$library:sound$vue_startup"). Make sure settings; device type (vaxstation 4000) and play settings (headphone jack) are selected. To play files from the DCL prompt (handy if you want to play sounds on a remote workstation) set a symbol up as follows;

```
PLAY == "$DECSOUND -VOLUME 50 -PLAY"
usage;
DCL> play sound.au
```

3) Audio port

The external audio port comes with a telephone-jack-like port. For starters, you can plug a telephone RECEIVER right into this port to hear your first sound files. After that, you can use the adapter (that came with the VaxStation), and plug in a small set of stereo speakers or headphones (the kind you'd plug into a WALKMAN, for example), for more volume. The adapter also has a microphone plug so that you can record sounds if DECsound is installed.

Playing audio files on micros

Most micros have at least a speaker built in, so theoretically all you need is the right software. Unfortunately most systems don't come bundled with sound-playing software, so there are many public domain or shareware software packages, each with their own bugs and features. Most separate sound recording hardware also comes with playing software, most of which can play sound (in the file format used by that hardware) even on machines that don't have that hardware installed.

PC or compatible

Chris S. Craig announces the following software for PCs:

```
ScopeTrax       This is a complete PC sound player/editor package.  Sounds
                can be played back at ANY rate between 1kHz to 65kHz through
                the PC speaker or the Sound Blaster.  It supports several
                file formats including VOC, IFF/8SVX, raw signed and raw
                unsigned.  A separate executable is provided to convert
                .au and mu-law to raw format.  ScopeTrax requires EGA/VGA
                graphics for editing and displaying sounds on a REALTIME
                oscilloscope.  The package also includes:
                      * An expanded memory player which can play sounds
                        larger than 640K in size.
                      * Basic (rough) sound compression/uncompression
                        utilities.
                      * Complete documentation.
                The package is FREEWARE!  It is available on SIMTEL in the
```

                PD1:[MSDOS.SOUND] directory.

One of the appendices below contains a list of more programs to play sound on the PC.

        Atari

For sounds on Atari STs - programs are in the atari/sound/players directory on atari.archive.umich.edu (141.211.164.8).

        Tandy

On a Tandy 1000, sounds can be played and recorded with DeskMate Sound (SOUND.PDM), or if they not stored in compressed format, they can also be played be a program called PLAYSND. No indication of whether PLAYSND is PD or not. It hasn’t been updated since March of 89.

        Amiga

On the Amiga, OmniPlay by David Champion
plays and converts IFF-8SVX, AIFF, WAV, VOC, .au, .snd, and 8 bit raw (signed, unsigned, u-law) samples. As of version 1.23, OmniPlay will also convert any playable sample to 8SVX. Files: wuarchive.wustl.edu
in /systems/amiga/audio/sampleplayers/oplay123.lha (?)
amiga.physik.unizh.ch in mus/play/oplay123.lha

        Apple Macintosh

Malcolm Slaney from Apple writes:

 "We do have tools to play sound back on most of our Unix hosts. We wrote a program called TcpPlay that lets us read a sound file on a Unix host, open a TCP/IP connection to the Mac on my desk, and plays the file. We think of it as X windows for sound (at least a step in that direction.)

 This software is available for anonymous FTP from ftp.apple.com
 [IP address 130.43.2.3 -- Guido].
 Look for ~ftp/pub/TcpPlay/TcpPlay.sit.hqx.

 Finally, there are MANY tools for working with sound on the Macintosh. Three applications that come to mind immediately are SoundEdit (formerly by Farralon and now by MacroMind/Paracomp), Alchemy and Eric Keller’s Signalyze. There are lots of other tools available for sound editing (including some of the QuickTime Movie tools.)"

Bill Houle sent the following lists:

EDITORS:

Sample Editor, Garrick McFarlane: plays AIFF and ’snd’ sounds. Can convert between AIFF and ’snd’. Can record from built-in mic. Can add effects such as fade, normalize, delay, etc.

Wavicle, Lee Fyock: plays SoundEdit files. Can convert to ’snd’. Can record from built-in mic. Can add effects such as fade, filter, reverb, etc.

[*]SoundEdit, Farallon: plays SoundEdit and ’snd’ sounds. Can read/write

SoundEdit files and 'snd' sounds. Can record from built-in mic. Can add effects such as echo, filter, reverb, etc.

PLAYERS:

The Player, Antoine Rosset & Mike Venturi: plays AIFF, SoundEdit, MOD, and 'snd' files.

SoundMaster (aka [*]Kaboom!), Bruce Tomlin: associates SoundEdit files to MacOS events.

SndControl, Riccardo Ettore: associates 'snd' sounds to MacOS events.

Most programs mentioned are shareware/freeware available from SUMEX and the various mirror sites, or check archie for the nearest FTP site. Popular commercial apps are indicated with a [*].

The Sound Site Newsletter

An electronic publication with lots of info about digitised sound and sound formats, albeit mostly on micros, is "The Sound Site Newsletter". Issue 12 appeared in March 1993. This used to be available by anonymouse ftp from saffron, but I have been informed that saffron is no longer providing this service. If you know of a different site carrying recent issues of the Sound Site newsletter, please let me know!)

Posting sounds

The newsgroup alt.binaries.sounds.misc is dedicated to postings containing sound. (Discussions related to such postings belong in alt.binaries.sounds.d.)

There is no set standard for posting sounds; uuencoded files in most popular formats are welcome, if split in parts under 50 kBytes. To accomodate automatic decoding software (such as the ":decode" command of the nn newsreader), please place a part indicator of the form (mm/nn) at the end of your subject meaning this is number mm of a total of nn part.

It is recommended to post sounds in the format that was used for the original recording; conversions to other formats often lose information and would do people with identical hardware as the poster no favor. For instance, convering 8-bit linear sound to U-LAW loses the lower few bits of the data, and rate changing conversions almost always add noise. Converting from U-LAW to linear requires expansion to 16 bit samples if no information loss is allowed!

U-LAW data is best posted with a NeXT/Sun header.

If you have to post a file in a headerless format (usually 8-bit linear, like ".snd"), please add a description giving at least the sampling rate and whether the bytes are signed (zero at 0) or unsigned (zero at 0200). However, it is highly recommended to add a header that indicates the sampling rate and encoding scheme; if necessary you can use SOX to add a header of your choice to raw data.

Compression of sound files usually isn’t worth it; the standard "compress" algorithm doesn’t save much when applied to sound data (typically at most 10-20 percent), and compression algorithms specifically designed for sound (e.g. NeXT’s) are usually proprietary. (See also the section "Compression schemes" earlier.)

Appendices

Here are some more detailed pieces of info that I received by e-mail.
They are reproduced here virtually without much editing.

Table of contents

FTP access for non-internet sites

From the sci.space FAQ:

> Sites not connected to the Internet cannot use FTP directly, but
> there are a few automated FTP servers which operate via email.
> Send mail containing only the word HELP to ftpmail@decwrl.dec.com
> or bitftp@pucc.princeton.edu, and the servers will send you
> instructions on how to make requests. (The bitftp service is no
> longer available through UUCP gateways due to complaints about
> overuse :-( )

Also:

> FAQ lists are available by anonymous FTP from rftm.mit.edu
> and by email from mail-server@rtfm.mit.edu (send a message
> containing "help" for instructions about the mail server).

AIFF Format (Audio IFF) and AIFC

This format was developed by Apple for storing high-quality sampled
sound and musical instrument info; it is also used by SGI and several
professional audio packages (sorry, I know no names). An extension,
called AIFC or AIFF-C, supports compression (see the last item below).

I’ve made a BinHex’ed MacWrite version of the AIFF spec (no idea if
it’s the same text as mentioned below) available by anonymous ftp from
ftp.cwi.nl [192.16.184.180]; the file is /pub/audio/AudioIFF1.2.hqx.
But you may be better off with the AIFF-C specs, see below.

Mike Brindley (brindley@ece.orst.edu) writes:

"The complete AIFF spec by Steve Milne, Matt Deatherage (Apple) is

available in ’AMIGA ROM Kernal Reference Manual: Devices (3rd Edition)’
1991 by Commodore-Amiga, Inc.; Addison-Wesley Publishing Co.;
ISBN 0-201-56775-X, starting on page 435 (this edition has a charcoal
grey cover). It is available in most bookstores, and soon in many
good librairies."

According to Mark Callow (msc@sgi.com):

A PostScript version of the AIFF-C specification is available via
anonymous ftp on FTP.SGI.COM (192.48.153.1) as /sgi/aiff-c.9.26.91.ps.

The NeXT/Sun audio file format

Here’s the complete story on the file format, from the NeXT
documentation. (Note that the "magic" number is ((int)0x2e736e64),
which equals ".snd".) Also, at the end, I’ve added a litte document
that someone posted to the net a couple of years ago, that describes
the format in a bit-by-bit fashion rather than from C.

I received this from Doug Keislar, NeXT Computer. This is also the
Sun format, except that Sun doesn’t recognize as many format codes. I
added the numeric codes to the table of formats and sorted it.

SNDSoundStruct: How a NeXT Computer Represents Sound

The NeXT sound software defines the SNDSoundStruct structure to
represent sound. This structure defines the soundfile and Mach-O
sound segment formats and the sound pasteboard type. It’s also used
to describe sounds in Interface Builder. In addition, each instance
of the Sound Kit’s Sound class encapsulates a SNDSoundStruct and
provides methods to access and modify its attributes.

Basic sound operations, such as playing, recording, and cut-and-paste
editing, are most easily performed by a Sound object. In many cases,
the Sound Kit obviates the need for in-depth understanding of the
SNDSoundStruct architecture. For example, if you simply want to
incorporate sound effects into an application, or to provide a simple
graphic sound editor (such as the one in the Mail application), you
needn’t be aware of the details of the SNDSoundStruct. However, if
you want to closely examine or manipulate sound data you should be
familiar with this structure.

The SNDSoundStruct contains a header, information that describes the
attributes of a sound, followed by the data (usually samples) that
represents the sound. The structure is defined (in
sound/soundstruct.h) as:

```
typedef struct {
    int magic;               /* magic number SND_MAGIC */
    int dataLocation;        /* offset or pointer to the data */
    int dataSize;            /* number of bytes of data */
    int dataFormat;          /* the data format code */
    int samplingRate;        /* the sampling rate */
    int channelCount;        /* the number of channels */
    char info[4];            /* optional text information */
} SNDSoundStruct;
```

SNDSoundStruct Fields

magic

magic is a magic number that’s used to identify the structure as a SNDSoundStruct. Keep in mind that the structure also defines the soundfile and Mach-O sound segment formats, so the magic number is also used to identify these entities as containing a sound.

dataLocation

It was mentioned above that the SNDSoundStruct contains a header followed by sound data. In reality, the structure only contains the header; the data itself is external to, although usually contiguous with, the structure. (Nonetheless, it’s often useful to speak of the SNDSoundStruct as the header and the data.) dataLocation is used to point to the data. Usually, this value is an offset (in bytes) from the beginning of the SNDSoundStruct to the first byte of sound data. The data, in this case, immediately follows the structure, so dataLocation can also be thought of as the size of the structure’s header. The other use of dataLocation, as an address that locates data that isn’t contiguous with the structure, is described in "Format Codes," below.

dataSize, dataFormat, samplingRate, and channelCount

These fields describe the sound data.

dataSize is its size in bytes (not including the size of the SNDSoundStruct).

dataFormat is a code that identifies the type of sound. For sampled sounds, this is the quantization format. However, the data can also be instructions for synthesizing a sound on the DSP. The codes are listed and explained in "Format Codes," below.

samplingRate is the sampling rate (if the data is samples). Three sampling rates, represented as integer constants, are supported by the hardware:

Constant	Sampling Rate (samples/sec)
SND_RATE_CODEC	8012.821 (CODEC input)
SND_RATE_LOW	22050.0 (low sampling rate output)
SND_RATE_HIGH	44100.0 (high sampling rate output)

channelCount is the number of channels of sampled sound.

info

info is a NULL-terminated string that you can supply to provide a textual description of the sound. The size of the info field is set when the structure is created and thereafter can't be enlarged. It's at least four bytes long (even if it's unused).

Format Codes

A sound's format is represented as a positive 32-bit integer. NeXT reserves the integers 0 through 255; you can define your own format and represent it with an integer greater than 255. Most of the formats defined by NeXT describe the amplitude quantization of sampled sound data:

Value	Code	Format
0	SND_FORMAT_UNSPECIFIED	unspecified format
1	SND_FORMAT_MULAW_8	8-bit mu-law samples
2	SND_FORMAT_LINEAR_8	8-bit linear samples
3	SND_FORMAT_LINEAR_16	16-bit linear samples
4	SND_FORMAT_LINEAR_24	24-bit linear samples
5	SND_FORMAT_LINEAR_32	32-bit linear samples
6	SND_FORMAT_FLOAT	floating-point samples
7	SND_FORMAT_DOUBLE	double-precision float samples
8	SND_FORMAT_INDIRECT	fragmented sampled data
9	SND_FORMAT_NESTED	?
10	SND_FORMAT_DSP_CORE	DSP program
11	SND_FORMAT_DSP_DATA_8	8-bit fixed-point samples
12	SND_FORMAT_DSP_DATA_16	16-bit fixed-point samples
13	SND_FORMAT_DSP_DATA_24	24-bit fixed-point samples
14	SND_FORMAT_DSP_DATA_32	32-bit fixed-point samples
15	?	
16	SND_FORMAT_DISPLAY	non-audio display data
17	SND_FORMAT_MULAW_SQUELCH	?
18	SND_FORMAT_EMPHASIZED	16-bit linear with emphasis
19	SND_FORMAT_COMPRESSED	16-bit linear with compression
20	SND_FORMAT_COMPRESSED_EMPHASIZED	A combination of the two above
21	SND_FORMAT_DSP_COMMANDS	Music Kit DSP commands
22	SND_FORMAT_DSP_COMMANDS_SAMPLES	?
[Some new ones supported by Sun. This is all I currently know. --GvR]		
23	SND_FORMAT_ADPCM_G721	
24	SND_FORMAT_ADPCM_G722	
25	SND_FORMAT_ADPCM_G723_3	
26	SND_FORMAT_ADPCM_G723_5	
27	SND_FORMAT_ALAW_8	

Most formats identify different sizes and types of sampled data. Some deserve special note:

--	SND_FORMAT_DSP_CORE format contains data that represents a loadable DSP core program. Sounds in this format are required by the SNDBootDSP() and SNDRunDSP() functions. You create a SND_FORMAT_DSP_CORE sound by reading a DSP load file (extension

".lod") with the SNDReadDSPfile() function.

--   SND_FORMAT_DSP_COMMANDS is used to distinguish sounds that contain DSP commands created by the Music Kit. Sounds in this format can only be created through the Music Kit's Orchestra class, but can be played back through the SNDStartPlaying() function.

--   SND_FORMAT_DISPLAY format is used by the Sound Kit's SoundView class. Such sounds can't be played.

--   SND_FORMAT_INDIRECT indicates data that has become fragmented, as described in a separate section, below.

--   SND_FORMAT_UNSPECIFIED is used for unrecognized formats.

Fragmented Sound Data

Sound data is usually stored in a contiguous block of memory. However, when sampled sound data is edited (such that a portion of the sound is deleted or a portion inserted), the data may become discontiguous, or fragmented. Each fragment of data is given its own SNDSoundStruct header; thus, each fragment becomes a separate SNDSoundStruct structure. The addresses of these new structures are collected into a contiguous, NULL-terminated block; the dataLocation field of the original SNDSoundStruct is set to the address of this block, while the original format, sampling rate, and channel count are copied into the new SNDSoundStructs.

Fragmentation serves one purpose: It avoids the high cost of moving data when the sound is edited. Playback of a fragmented sound is transparent-you never need to know whether the sound is fragmented before playing it. However, playback of a heavily fragmented sound is less efficient than that of a contiguous sound. The SNDCompactSamples() C function can be used to compact fragmented sound data.

Sampled sound data is naturally unfragmented. A sound that's freshly recorded or retrieved from a soundfile, the Mach-O segment, or the pasteboard won't be fragmented. Keep in mind that only sampled data can become fragmented.

____
>From mentor.cc.purdue.edu!purdue!decwrl!ucbvax!ziploc!eps Wed Apr 4 23:56:23 EST 1990
Article 5779 of comp.sys.next:
Path: mentor.cc.purdue.edu!purdue!decwrl!ucbvax!ziploc!eps
>From: eps@toaster.SFSU.EDU (Eric P. Scott)
~Newsgroups: comp.sys.next
~Subject: Re: Format of NeXT sndfile headers?
Message-ID: <445@toaster.SFSU.EDU>
~Date: 31 Mar 90 21:36:17 GMT
~References: <14978@phoenix.Princeton.EDU>

```
~Reply-To: eps@cs.SFSU.EDU (Eric P. Scott)
Organization: San Francisco State University
~Lines: 42

In article <14978@phoenix.Princeton.EDU>
        bskendig@phoenix.Princeton.EDU (Brian Kendig) writes:
>I'd like to take a program I have that converts Macintosh sound
files
>to NeXT sndfiles and polish it up a bit to go the other direction as
>well.

Two people have already submitted programs that do this
(Christopher Lane and Robert Hood); check the various
NeXT archive sites.

>       Could someone please give me the format of a NeXT sndfile
>header?

"big-endian"
        0       1       2       3
        +-------+-------+-------+-------+
0       | 0x2e  | 0x73  | 0x6e  | 0x64  |       "magic" number
        +-------+-------+-------+-------+
4       |                               |       data location
        +-------+-------+-------+-------+
8       |                               |       data size
        +-------+-------+-------+-------+
12      |                               |       data format (enum)
        +-------+-------+-------+-------+
16      |                               |       sampling rate (int)
        +-------+-------+-------+-------+
20      |                               |       channel count
        +-------+-------+-------+-------+
24      |       |       |       |       |       (optional) info
string

28 = minimum value for data location

data format values can be found in /usr/include/sound/soundstruct.h

Most common combinations:

         sampling  channel    data
             rate    count  format
voice file   8012        1       1 =  8-bit mu-law
system beep 22050        2       3 = 16-bit linear
CD-quality  44100        2       3 = 16-bit linear

------------------------------------------------------------------------
IFF/8SVX Format
---------------

~Newsgroups: alt.binaries.sounds.d,alt.sex.sounds
~Subject: Format of the IFF header (Amiga sounds)
Message-ID: <2509@tardis.Tymnet.COM>
~From: jms@tardis.Tymnet.COM (Joe Smith)
~Date: 23 Oct 91 23:54:38 GMT
Followup-To: alt.binaries.sounds.d
Organization: BT North America (Tymnet)

The first 12 bytes of an IFF file are used to distinguish between an Amiga
```

picture (FORM-ILBM), an Amiga sound sample (FORM-8SVX), or other file
conforming to the IFF specification. The middle 4 bytes is the count of
bytes that follow the "FORM" and byte count longwords. (Numbers are stored
in M68000 form, high order byte first.)

FutureSound audio file, 15000 samples at 10.000KHz, file is 15048 bytes long.

```
0000: 464F524D 00003AC0 38535658 56484452    FORM..:.8SVXVHDR
      F O R M     15040 8 S V X  V H D R
0010: 00000014 00003A98 00000000 00000000    ......:.........
            20    15000        0        0
0020: 27100100 00010000 424F4459 00003A98    '.......BODY..:.
     10000 1 0    1.0   B O D Y     15000
```

```
0000000..03 = "FORM", identifies this as an IFF format file.
FORM+00..03 (ULONG) = number of bytes that follow.  (Unsigned long int.)
FORM+03..07 = "8SVX", identifies this as an 8-bit sampled voice.

????+00..03 = "VHDR", Voice8Header, describes the parameters for the BODY.
VHDR+00..03 (ULONG) = number of bytes to follow.
VHDR+04..07 (ULONG) = samples in the high octave 1-shot part.
VHDR+08..0B (ULONG) = samples in the high octave repeat part.
VHDR+0C..0F (ULONG) = samples per cycle in high octave (if repeating), else 0.
VHDR+10..11 (UWORD) = samples per second.  (Unsigned 16-bit quantity.)
VHDR+12     (UBYTE) = number of octaves of waveforms in sample.
VHDR+13     (UBYTE) = data compression (0=none, 1=Fibonacci-delta encoding).
VHDR+14..17 (FIXED) = volume.  (The number 65536 means 1.0 or full volume.)

????+00..03 = "BODY", identifies the start of the audio data.
BODY+00..03 (ULONG) = number of bytes to follow.
BODY+04..NNNNN      = Data, signed bytes, from -128 to +127.
```

```
0030: 04030201 02030303 04050605 05060605
0040: 06080806 07060505 04020202 01FF0000
0050: 00000000 FF00FFFF FFFEFDFD FDFEFFFF
0060: FDFDFF00 00FFFFFF 00000000 00FFFF00
0070: 00000000 00FF0000 00FFFEFF 00000000
0080: 00010000 000101FF FF0000FE FEFFFFFE
0090: FDFDFEFD FDFFFFFC FDFEFDFD FEFFFEFE
00A0: FFFEFEFE FEFEFEFF FFFFFEFF 00FFFF01
```

This small section of the audio sample shows the number ranging from -5 (0xFD)
to +8 (0x08). Warning: Do not assume that the BODY starts 48 bytes into the
file. In addition to "VHDR", chunks labeled "NAME", "AUTH", "ANNO", or
"(c) " may be present, and may be in any order. You will have to check the
byte count in each chunk to determine how many bytes to skip.

Playing sound on a PC

~From: Eric A Rasmussen

Any turbo PC (8088 at 8 Mhz or greater)/286/386/486/etc. can produce a quality
playback of single channel 8 bit sounds on the internal (1 bit, 1 channel)
speaker by utilizing Pulse-Width-Modulation, which toggles the speaker faster
than it can physically move to simulate positions between fully on and fully
off. There are several PD programs of this nature that I know of:

```
REMAC  - Plays MAC format sound files.  Files on the Macintosh, at least the
         sound files that I've ripped apart, seem to contain 3 parts.  The
         first two are info like what the file icon looks like and other
         header type info.  The third part contains the raw sample data, and
         it is this portion of the file which is saved to a seperate file,
         often named with the .snd extension by PC users.  Personally, I like
         to name the files .s1, .s2, .s3, or .s4 to indicate the sampling rate
         of the file. (-s# is how to specify the playback rate in REMAC.)
         REMAC provides playback rates of 5550hz, 7333hz, 11 khz, & 22 khz.
REMAC2 - Same as REMAC, but sounds better on higher speed machines.
REPLAY - Basically same as REMAC, but for playback of Atari ST sounds.
         Apparently, the Atari has two sound formats, one of which sounds like
         garbage if played by REMAC or REPLAY in the incorrect mode.  The
         other file format works fine with REMAC and so appears to be 'normal'
         unsigned 8-bit data.  REPLAY provides playback rates of 11.5 khz,
         12.5 khz, 14 khz, 16 khz, 18.5 khz, 22khz, & 27 khz.
```

These three programs are all by the same author, Richard E. Zobell who does not have an internet mail address to my knowledge, but does have a GEnie email address of R.ZOBELL.

Additionally, there are various stand-alone demos which use the internal speaker, of which there is one called mushroom which plays a 30 second advertising jingle for magic mushroom room deoderizers which is pretty humerous. I've used this player to playback samples that I ripped out of the commercial game program Mean Streets, which uses something they call RealSound (tm) to playback digital samples on the internal speaker. (Of course, I only do this on my own system, and since I own the game, I see no problems with it.)

For owners of 8 Mhz 286's and above, the option to play 4 channel 8 bit sounds (with decent quality) on the internal speaker is also a reality. Quite a number of PD programs exist to do this, including, but not limited to:

ModEdit, ModPlay, ScreamTracker, STM, Star Trekker, Tetra, and probably a few more.

All these programs basically make use of various sound formats used by the Amiga line of computers. These include .stm files, .mod files [a.k.a. mod. files], and .nst files [really the same hing]. Also, these programs pretty much all have the option to playback the sound to add-on hardware such as the SoundBlaster card, the Covox series of devices, and also to direct the data to either one or two (for stereo) parallel ports, which you could attach your own D/A's to. (From what I have seen, the Covox is basically an small amplified speaker with a D/A which plugs into the parallel port. This sounds very similiar to the Disney Sound System (DSS) which people have been talking about recently.)

The EA-IFF-85 documentation

~From: dgc3@midway.uchicago.edu

As promised, here's an ftp location for the EA-IFF-85 documentation. It's the November 1988 release as revised by Commodore (the last public release), with specifications for IFF FORMs for graphics, sound, formatted text, and more. IFF FORMS now exist for other media, including structured drawing, and new documentation is now available only from Commodore.

The documentation is at grind.isca.uiowa.edu [128.255.19.233], in the directory /amiga/f1/ff185. The complete file list is as follows:

DOCUMENTS.zoo
EXAMPLES.zoo
EXECUTABLE.zoo
INCLUDE.zoo
LINKER_INFO.zoo
OBJECT.zoo
SOURCE.zoo
TP_IFF_Specs.zoo

All files except DOCUMENTS.zoo are Amiga-specific, but may be used as a basis for conversion to other platforms. Well, I take that tentatively back. I don't know what TP_IFF_Specs.zoo contains, so it might be non-Amiga-specific.

US Federal Standard 1016 availability

~From: jpcampb@afterlife.ncsc.mil (Joe Campbell)

The U.S. DoD's Federal-Standard-1016 based 4800 bps code excited linear prediction voice coder version 3.2 (CELP 3.2) Fortran and C simulation source codes are available for worldwide distribution (on DOS diskettes, but configured to compile on Sun SPARC stations) from NTIS and DTIC. Example input and processed speech files are included. A Technical Information Bulletin (TIB), "Details to Assist in Implementation of Federal Standard 1016 CELP," and the official standard, "Federal Standard 1016, Telecommunications: Analog to Digital Conversion of Radio Voice by 4,800 bit/second Code Excited Linear Prediction (CELP)," are also available.

This is available through the National Technical Information Service:

NTIS
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
USA
(703) 487-4650

The "AD" ordering number for the CELP software is AD M000 118 (US$ 90.00) and for the TIB it's AD A256 629 (US$ 17.50). The LPC-10 standard, described below, is FIPS Pub 137 (US$ 12.50). There is a $3.00 shipping charge on all U.S. orders. The telephone number for their automated system is 703-487-4650, or 703-487-4600 if you'd prefer to talk with a real person.

(U.S. DoD personnel and contractors can receive the package from the Defense Technical Information Center: DTIC, Building 5, Cameron Station, Alexandria, VA 22304-6145. Their telephone number is 703-274-7633.)

The following articles describe the Federal-Standard-1016 4.8-kbps CELP coder (it's unnecessary to read more than one):

Campbell, Joseph P. Jr., Thomas E. Tremain and Vanoy C. Welch, "The Federal Standard 1016 4800 bps CELP Voice Coder," Digital Signal Processing, Academic Press, 1991, Vol. 1, No. 3, p. 145-155.

Campbell, Joseph P. Jr., Thomas E. Tremain and Vanoy C. Welch, "The DoD 4.8 kbps Standard (Proposed Federal Standard 1016)," in Advances in Speech Coding, ed. Atal, Cuperman and Gersho, Kluwer Academic Publishers, 1991, Chapter 12, p. 121-133.

Campbell, Joseph P. Jr., Thomas E. Tremain and Vanoy C. Welch, "The Proposed Federal Standard 1016 4800 bps Voice Coder: CELP," Speech Technology Magazine, April/May 1990, p. 58-64.

The U.S. DoD's Federal-Standard-1015/NATO-STANAG-4198 based 2400 bps linear prediction coder (LPC-10) was republished as a Federal Information Processing Standards Publication 137 (FIPS Pub 137).
It is described in:

Thomas E. Tremain, "The Government Standard Linear Predictive Coding Algorithm: LPC-10," Speech Technology Magazine, April 1982, p. 40-49.

There is also a section about FS-1015 in the book:
Panos E. Papamichalis, Practical Approaches to Speech Coding, Prentice-Hall, 1987.

The voicing classifier used in the enhanced LPC-10 (LPC-10e) is described in:
Campbell, Joseph P., Jr. and T. E. Tremain, "Voiced/Unvoiced Classification of Speech with Applications to the U.S. Government LPC-10E Algorithm," Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, 1986, p. 473-6.

Copies of the official standard
"Federal Standard 1016, Telecommunications: Analog to Digital Conversion of Radio Voice by 4,800 bit/second Code Excited Linear Prediction (CELP)"
are available for US$ 5.00 each from:

GSA Federal Supply Service Bureau
Specification Section, Suite 8100
470 E. L'Enfant Place, S.W.
Washington, DC 20407
(202)755-0325

Realtime DSP code for FS-1015 and FS-1016 is sold by:

John DellaMorte
DSP Software Engineering
165 Middlesex Tpk, Suite 206
Bedford, MA 01730
USA
1-617-275-3733
1-617-275-4323 (fax)
dspse.bedford@channel1.com

DSP Software Engineering's FS-1016 code can run on a DSP Research's Tiger 30
(a PC board with a TMS320C3x and analog interface suited to development work).

DSP Research
1095 E. Duane Ave.
Sunnyvale, CA 94086
USA
(408)773-1042
(408)736-3451 (fax)

~From: tobiasr@monolith.lrmsc.loral.com (Richard Tobias)

For U.S. FED-STD-1016 (4800 bps CELP) _realtime_ DSP code and
information about products using this code using the AT&T DSP32C and

```
AT&T DSP3210, contact:

White Eagle Systems Technology, Inc.
1123 Queensbridge Way
San Jose, CA 95120
(408) 997-2706
(408) 997-3584 (fax)
rjjt@netcom.com

~From: Cole Erskine <cole@analogical.com>

[paraphrased]

Analogical Systems has a _real-time_ multirate implementation of U.S.
Federal Standard 1016 CELP operating at bit rates of 4800, 7200, and
9600 bps on a single 27MHz Motorola DSP56001. Source and object code
is available for a one-time license fee.

FREE, _real-time_ demonstration software for the Ariel PC-56D is
available for those who already have such a board by contacting
Analogical Systems.  The demo software allows you to record and
playback CELP files to and from the PC’s hard disk.

Analogical Systems
2916 Ramona Street
Palo Alto, CA 94306
Tel: +1 (415) 323-3232
FAX: +1 (415) 323-4222

------------------------------------------------------------------------
Creative Voice (VOC) file format
--------------------------------

~From: galt@dsd.es.com

(byte numbers are hex!)

    HEADER (bytes 00-19)
    Series of DATA BLOCKS (bytes 1A+) [Must end w/ Terminator Block]

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

HEADER:
=======
     byte #     Description
     ------     ------------------------------------------
     00-12      "Creative Voice File"
     13         1A (eof to abort printing of file)
     14-15      Offset of first datablock in .voc file (std 1A 00
                in Intel Notation)
     16-17      Version number (minor,major) (VOC-HDR puts 0A 01)
     18-19      2’s Comp of Ver. # + 1234h (VOC-HDR puts 29 11)

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

DATA BLOCK:
===========

   Data Block:  TYPE(1-byte), SIZE(3-bytes), INFO(0+ bytes)
   NOTE: Terminator Block is an exception -- it has only the TYPE byte.
```

TYPE	Description	Size (3-byte int)	Info
00	Terminator	(NONE)	(NONE)
01	Sound data	2+length of data	*
02	Sound continue	length of data	Voice Data
03	Silence	3	**
04	Marker	2	Marker# (2 bytes)
05	ASCII	length of string	null terminated string
06	Repeat	2	Count# (2 bytes)
07	End repeat	0	(NONE)
08	Extended	4	***

*Sound Info Format:

00	Sample Rate
01	Compression Type
02+	Voice Data

**Silence Info Format:

00-01	Length of silence - 1
02	Sample Rate

```
***Extended Info Format:
   ---------------------
   00-01  Time Constant: Mono: 65536 - (256000000/sample_rate)
                         Stereo: 65536 - (25600000/(2*sample_rate))
   02     Pack
   03     Mode: 0 = mono
                1 = stereo


Marker#            -- Driver keeps the most recent marker in a status byte
Count#             -- Number of repetitions + 1
                        Count# may be 1 to FFFE for 0 - FFFD repetitions
                        or FFFF for endless repetitions
Sample Rate        -- SR byte = 256-(1000000/sample_rate)
Length of silence -- in units of sampling cycle
Compression Type  -- of voice data
                        8-bits    = 0
                        4-bits    = 1
                        2.6-bits  = 2
                        2-bits    = 3
                        Multi DAC = 3+(# of channels) [interesting--
                                      this isn't in the developer's manual]
```

RIFF WAVE (.WAV) file format

RIFF is a format by Microsoft and IBM which is similar in spirit and functionality as EA-IFF-85, but not compatible (and it's in little-endian byte order, of course :-). WAVE is RIFF's equivalent of AIFF, and its inclusion in Microsoft Windows 3.1 has suddenly made it important to know about.

Rob Ryan was kind enough to send me a description of the RIFF format. Unfortunately, it is too big to include here (27 k), but I've made it available for anonymous ftp as ftp.cwi.nl:/pub/audio/RIFF-format.

And here's a pointer to the official description from Matt Saettler, Microsoft Multimedia:

"The complete definition of the WAVE file format as defined by IBM/Microsoft is available for anon. FTP from ftp.uu.net in the vendor/microsoft/multimedia directory."

(Rob Ryan’s version may actually be an extract from one of the files
stored there.)

U-LAW and A-LAW definitions

[Adapted from information provided by duggan@cc.gatech.edu (Rick
Duggan) and davep@zenobia.phys.unsw.EDU.AU (David Perry)]

u-LAW (really mu-LAW) is

```
          sgn(m)   (     |m |)       |m |
   y=    ------- ln( 1+ u|--|)       |--| =< 1
         ln(1+u)   (     |mp|)       |mp|
```

A-LAW is

```
     |     A    (m )                |m |    1
     |  ------- (--)                |--| =< -
     |  1+ln A  (mp)                |mp|    A
   y=|
     | sgn(m) (        |m |)    1    |m |
     | ------ ( 1+ ln A|--|)    - =< |--| =< 1
     | 1+ln A (        |mp|)    A    |mp|
```

Values of u=100 and 255, A=87.6, mp is the Peak message value, m is
the current quantised message value. (The formulae get simpler if you
substitute x for m/mp and sgn(x) for sgn(m); then -1 <= x <= 1.)

Converting from u-LAW to A-LAW is in a sense "lossy" since there are
quantizing errors introduced in the conversion.

"..the u-LAW used in North America and Japan, and the
A-LAW used in Europe and the rest of the world and
international routes.."

~References:

Modern Digital and Analog Communication Systems, B.P.Lathi., 2nd ed.
ISBN 0-03-027933-X

Transmission Systems for Communications
Fifth Edition
by Members of the Technical Staff at Bell Telephone Laboratories
Bell Telephone Laboratories, Incorporated
Copyright 1959, 1964, 1970, 1982

A note on the resolution of U-LAW by Frank Klemm <pfk@rz.uni-jena.de>:

8 bit U-LAW has the same lowest magnitude like 12 bit linear and 12 bit
U-LAW like 16 linear.

Device/Coding	Resolution	Resolution	
	on maximal level	on low level	
8 bit linear	8	8	
8 bit ulaw	6	12	(used for digital telephone)
12 bit linear	12	12	
12 bit ulaw	10	16	(used in DAT/Longplay)
16 bit linear	16	16	

```
estimated for some analoge technique:
tape recorder (HiFi DIN)
                 8                        9      (no Problem today)
tape recorder (semiprofessional)
                10.5                    13.5

------------------------------------------------------------------------
AVR File Format
---------------
```

~From: hyc@hanauma.Jpl.Nasa.Gov (Howard Chu)

A lot of PD software exists to play Mac .snd files on the ST. One other
format that seems pretty popular (used by a number of commercial packages)
is the AVR format (from Audio Visual Research). This format has a 128 byte
header that looks like this:

```
        char magic[4]="2BIT";
        char name[8];           /* null-padded sample name */
        short mono;             /* 0 = mono, 0xffff = stereo */
        short rez;              /* 8 = 8 bit, 16 = 16 bit */
        short sign;             /* 0 = unsigned, 0xffff = signed */
        short loop;             /* 0 = no loop, 0xffff = looping sample */
        short midi;             /* 0xffff = no MIDI note assigned,
                                   0xffXX = single key note assignment
                                   0xLLHH = key split, low/hi note */
        long rate;              /* sample frequency in hertz */
        long size;              /* sample length in bytes or words (see rez) */
        long lbeg;              /* offset to start of loop in bytes or words.
                                   set to zero if unused. */
        long lend;              /* offset to end of loop in bytes or words.
                                   set to sample length if unused. */
        short res1;             /* Reserved, MIDI keyboard split */
        short res2;             /* Reserved, sample compression */
        short res3;             /* Reserved */
        char ext[20];           /* Additional filename space, used
                                   if (name[7] != 0) */
        char user[64];          /* User defined. Typically ASCII message. */
```

```
-----------------------------------------------------------------------
The Amiga MOD Format
--------------------
```

~From: norlin@mailhost.ecn.uoknor.edu (Norman Lin)

MOD files are music files containing 2 parts:

(1) a bank of digitized samples
(2) sequencing information describing how and when to play the samples

MOD files originated on the Amiga, but because of their flexibility
and the extremely large number of MOD files available, MOD players
are now available for a variety of machines (IBM PC, Mac, Sparc
Station, etc.)

The samples in a MOD file are raw, 8 bit, signed, headerless, linear
digital data. There may be up to 31 distinct samples in a MOD file,
each with a length of up to 128K (though most are much smaller; say,
10K - 60K). An older MOD format only allowed for up to 15 samples in
a MOD file; you don’t see many of these anymore. There is no standard
sampling rate for these samples. [But see below.]

```
The sequencing information in a MOD file contains 4 tracks of
information describing which, when, for how long, and at what frequency
samples should be played.  This means that a MOD file can have up
to 31 distinct (digitized) instrument sounds, with up to 4 playing
simultaneously at any given point.  This allows a wide variety
of orchestrational possibilities, including use of voice samples
or creation of one’s own instruments (with appropriate sampling
hardware/software).  The ability to use one’s own samples as instruments
is a flexibility that other music files/formats do not share, and
is one of the reasons MOD files are so popular, numerous, and diverse.

15 instrument MODs, as noted above, are somewhat older than 31
instrument MODs and are not (at least not by me) seen very often
anymore.  Their format is identical to that of 31 instrument MODs
except:

(1) Since there are only 15 samples, the information for the last (15th)
    sample starts at byte 440 and goes through byte 469.
(2) The songlength is at byte 470 (contrast with byte 950 in 31 instrument
    MOD)
(3) Byte 471 appears to be ignored, but has been observed to be 127.
    (Sorry, this is from observation only)
(4) Byte 472 begins the pattern sequence table (contrast with byte 952
    in a 31 instrument MOD)
(5) Patterns start at byte 600 (contrast with byte 1084 in 31 instrument MOD)

"ProTracker," an Amiga MOD file creator/editor, is available for ftp
everywhere as pt??.lzh.

~From: Apollo Wong <apollo@ee.ualberta.ca>

~From: M.J.H.Cox@bradford.ac.uk (Mark Cox)
~Newsgroups: alt.sb.programmer
~Subject: Re: Format for MOD files...
Message-ID: <1992Mar18.103608.4061@bradford.ac.uk>
~Date: 18 Mar 92 10:36:08 GMT
Organization: University of Bradford, UK

wdc50@DUTS.ccc.amdahl.com (Winthrop D Chan) writes:
>I’d like to know if anyone has a reference document on the format of the
>Amiga Sound/NoiseTracker (MOD) files. The author of Modplay said he was going
>to release such a document sometime last year, but he never did. If anyone

I found this one, which covers it better than I can explain it - if you
use this in conjunction with the documentation that comes with Norman
Lin’s Modedit program it should pretty much cover it.

Mark J Cox

/***********************************************************************

Protracker 1.1B Song/Module Format:
-----------------------------------

Offset  Bytes  Description
------  -----  -----------
   0     20    Songname. Remember to put trailing null bytes at the end...

Information for sample 1-31:

Offset  Bytes  Description
```

```
------  -----  -----------
  20     22    Samplename for sample 1. Pad with null bytes.
  42      2    Samplelength for sample 1. Stored as number of words.
               Multiply by two to get real sample length in bytes.
  44      1    Lower four bits are the finetune value, stored as a signed
               four bit number. The upper four bits are not used, and
               should be set to zero.
               Value:  Finetune:
                 0        0
                 1       +1
                 2       +2
                 3       +3
                 4       +4
                 5       +5
                 6       +6
                 7       +7
                 8       -8
                 9       -7
                 A       -6
                 B       -5
                 C       -4
                 D       -3
                 E       -2
                 F       -1

  45      1    Volume for sample 1. Range is $00-$40, or 0-64 decimal.
  46      2    Repeat point for sample 1. Stored as number of words offset
               from start of sample. Multiply by two to get offset in bytes.
  48      2    Repeat Length for sample 1. Stored as number of words in
               loop. Multiply by two to get replen in bytes.

Information for the next 30 samples starts here. It’s just like the info for
sample 1.

Offset  Bytes  Description
------  -----  -----------
  50     30    Sample 2...
  80     30    Sample 3...
   .
   .
   .
 890     30    Sample 30...
 920     30    Sample 31...

Offset  Bytes  Description
------  -----  -----------
 950      1    Songlength. Range is 1-128.
 951      1    Well... this little byte here is set to 127, so that old
               trackers will search through all patterns when loading.
               Noisetracker uses this byte for restart, but we don’t.
 952    128    Song positions 0-127. Each hold a number from 0-63 that
               tells the tracker what pattern to play at that position.
1080      4    The four letters "M.K." - This is something Mahoney & Kaktus
               inserted when they increased the number of samples from
               15 to 31. If it’s not there, the module/song uses 15 samples
               or the text has been removed to make the module harder to
               rip. Startrekker puts "FLT4" or "FLT8" there instead.

Offset  Bytes  Description
------  -----  -----------
1084    1024   Data for pattern 00.
```

```
   .
   .
   .
xxxx  Number of patterns stored is equal to the highest patternnumber
      in the song position table (at offset 952-1079).

Each note is stored as 4 bytes, and all four notes at each position in
the pattern are stored after each other.

00 -  chan1  chan2  chan3  chan4
01 -  chan1  chan2  chan3  chan4
02 -  chan1  chan2  chan3  chan4
etc.

Info for each note:

 _____byte 1_____   byte2_    _____byte 3_____   byte4_
/                \ /      \  /                \ /      \
0000          0000-00000000  0000          0000-00000000

Upper four    12 bits for    Lower four    Effect command.
bits of sam-  note period.   bits of sam-
ple number.                  ple number.

Periodtable for Tuning 0, Normal
  C-1 to B-1 : 856,808,762,720,678,640,604,570,538,508,480,453
  C-2 to B-2 : 428,404,381,360,339,320,302,285,269,254,240,226
  C-3 to B-3 : 214,202,190,180,170,160,151,143,135,127,120,113

To determine what note to show, scan through the table until you find
the same period as the one stored in byte 1-2. Use the index to look
up in a notenames table.

This is the data stored in a normal song. A packed song starts with the
four letters "PACK", but i don't know how the song is packed: You can
get the source code for the cruncher/decruncher from us if you need it,
but I don't understand it; I've just ripped it from another tracker...

In a module, all the samples are stored right after the patterndata.
To determine where a sample starts and stops, you use the sampleinfo
structures in the beginning of the file (from offset 20). Take a look
at the mt_init routine in the playroutine, and you'll see just how it
is done.

Lars "ZAP" Hamre/Amiga Freelancers

***********************************************************************/

--
Mark J Cox -----
Bradford, UK ---


PS: A file with even *much* more info on MOD files, compiled by Lars
Hamre, is available from ftp.cwi.nl:/pub/audio/MOD-info.  Enjoy!


FTP sites for MODs and MOD players
----------------------------------

~Subject: MODS AND PLAYERS!! **READ** info/where to get them
```

```
~From: cjohnson@tartarus.uwa.edu.au (Christopher Johnson)
~Newsgroups: alt.binaries.sounds.d
Message-ID: <1h32ivINNglu@uniwa.uwa.edu.au>
~Date: 21 Dec 92 00:19:43 GMT
Organization: The University of Western Australia

Hello world,

For all those asking, here is where to get those mod players and mods.

SNAKE.MCS.KENT.EDU is the best site for general stuff.  look in /pub/SB-Adlib

Simtel-20 or archie.au(simtel mirror) in <msdos.sound>

for windows players ftp.cica.indiana.edu in pub/pc/win3/sound

here is a short list of players

mp or modplay   BEST OVERALL                    mp219b.zip
        simtel and snake

wowii           best for vga/fast machines      wowii12b.zip
        simtel and snake

trakblaster     best for compatability          trak-something
        simtel and snake        two versions, old one for slow
        machines

ss              cute display(hifi)              have_sex.arj
        found on local BBS (western Australia White Ghost)

superpro player generally good                  ssp.zip or similar
        found on night owl 7 CD

player?         cute display(hifi)              player.zip or similar
        found on night owl 7 CD

WINDOWS

Winmod pro      does protracker                 wmp????.zip
        cica

winmod          more stable                     winmod12.zip or similar
        cica

Hope this helps, e-mail me if you find any more players and I will add them in for t
little out of hand.

for mods ftp to wuarchive.wustl.edu and go to the amiga music directory (pub/amiga/m
a while

see you soon

Chris.

-----------------------------------------------------------------------
```

Audio-Burner.com

MP3 Glossary - Audio and MP3 Terms

AIFF	Audio Interface File Format (AIFF) was developed by Apple and is the audio format for Macintosh computers. This format does not support compression.
Bit Rate	A bit rate is the amount of information (or bits) that is transferred per second (bit per second or bps). MP3s are measured in thousands of bits per second (kbps) and the higher the kbps, the higher the sound quality. 128 kbps is the standard MP3 bit rate.
Blade / BladeEnc	An open source WAV to MP3 encoding engine used in many MP3 software programs.
Buffer	An area of memory used to speed up access to devices such as a CD-R/W. It is used for temporary storage of data read from or waiting to be written to a CD-R/W.
Buffer Underrun	When a CD burner requests data from the write buffer and there is no data, the write laser doesn?t have any more data to write therefore causing a buffer underun. The CD recording cannot be interrupted in mid

Acoustica MP3 Software

:: MP3 CD Burner
:: MP3 to Wave Converter
:: MP3 Audio Mixer
:: CD Label Maker

MP3 Converter Software

:: Audio Convert
:: Advanced WMA Workshop

MP3 CD Burner Software

:: Data & MP3 CD Burner
:: MP3 CD Converter
:: X2CD Music CD Burner

CD Ripper Software

:: MP3 CD Extractor
:: AltoMP3 Maker
:: MP3 Master

MP3 Editors and Sound Recorders

:: Coding Workshop Ringtone Converter

	session and when this occurs you will have a defective CD burn.
Burn-Proof	Burn-Proof is a technology developed by Sanyo, which helps prevent buffer underruns.
CDDB	An online based Compact Disc Database (CDDB) that allows you to download artist names, album titles, track names etc. This information is displayed in your MP3 player.
CD-R	Compact Disc Recordable (CD-R) is a recording device that allows you to record data and audio CDs only one time.
CD-RW	Compact Disc Rewritable is an extension of CD-R whereby you can rewrite data or audio to the same CD multiple times.
CD-ROM	A Compact Disc-Read Only Memory (CD-ROM) is designed to store data in the form of text, graphics and audio. CD-ROMs use the Yellow Book standard as published by Philips.
CD Text	A red book CD standard that allows album titles, artist names, and song titles to be embedded on an audio CD. Some CD players can read this data.
Codec	A codec (compression/decompression) is any technology that is used for compressing and decompressing data such as audio (MP3) or video (MPEG).
Constant bit rate (CBR)	Constant Bit Rate is an encoding method that maintains the same bit rate across the entire audio file.
Digital audio extraction (DAE)	Digital Audio Extraction (DAE), also known as CD ripping, is the process of extracting audio from

DVD Software
:: DVD X Copy! :: DVD Squeeze :: DVD Echo :: Super DVD Ripper

	CD which is copied to hard drive in WAV format.
Decoder	The process of converting an MP3 to WAV. This may be done in order to edit your WAV file.
Disc at once (DAO)	During the CD burning process, the entire CD is written at once without the write laser being turned off.
Encoder	The process of converting a WAV to MP3 audio format. This reduces file size while maintaining sound quality.
ID3	A tagging system that allows you to put music information such as artist, song title, album title, lyrics within your audio files.
ISO	International Organization for Standardization (ISO) is a non-governmental global organization established in 1947 that works to develop standards across goods and services.
ISO 9660	A widely used data interchange format adopted in 1987. CDs created in this format can be read by Unix, Macintosh and Windows computers.
Jitter	Jitter is caused by the inability of many CD-ROM drives to accurately seek a specific sector on an audio CD therefore resulting in pops or clicks when listening to the ripped audio track.
LAME	**L**Ain't an **M**p3 **E**ncoder (LAME) is an open source MP3 encoder engine used in a large number of MP3 software titles.
MP3	MP3 is an acronym for MPEG layer 3, which is a compressed audio format. A compression ratio of up to 12 to 1 compression is possible, which produces high

	sound quality.	
MPEG	The Moving Picture Experts Group (MPEG) is a working group under the International Organization for Standardization (ISO) that sets the standards for encoding audio and video in digital format. Official MPEG Homepage	
Normalization	Due to volume differences between audio tracks, normalization allows you to set the volume consistent between tracks during the encoding or burning process.	
Ogg Vorbis	A non-proprietary, patent free open source audio compression format that is similar to MP3.	
Peer to Peer (P2P)	A sharing and delivery of user specified files among groups of people who are logged on to a file sharing network. Napster was the first mainstream P2P software that enabled large scale file sharing.	
Red Book CD	A CD audio standard defined by Sony and Philips in 1980 that was published in a red binder, hence the name.	
Ripping (CD Ripping)	The process of digitally extracting audio from CDs to your hard drive in WAV format.	
Streaming	Streaming audio allows for on-the-fly listening to an audio file. The audio file is streamed from a server where it is received and stored in your buffer on your computer. If you use WinAmp or RealAudio, you will see a message displayed telling you that the audio file is being buffered. The file is not saved on your hard drive.	
Track at once (TAO)	During the CD burning process, each track is written on its own and the writer laser is turned off before starting to burn the next	

	track. This causes a 2 second gap between tracks.
Variable bit rate (VBR)	Variable Bit Rate adjusts the bit rate depending on the sound. For example, if there is silence in a song, the bit rate would decrease and if there is a full symphonic sound the bit rate would increase. VBR offers a higher sound quality at a smaller file size.
Wave (.wav)	A file format for storing digital audio data in waveform.
WMA	Window's Media Audio is Microsoft's audio encoding format that is starting to gain popularity due to its high quality output at lower file sizes. A 96 kbps (and in some tests a 64 kbps) wma file is equivalent in sound quality to a 128 kbps MP3 file.
Yellow Book	A CD-ROM standard defined by Sony and Philips in 1983 that is an extension of the Red Book standard. This standard enables CDs to contain data along with audio.

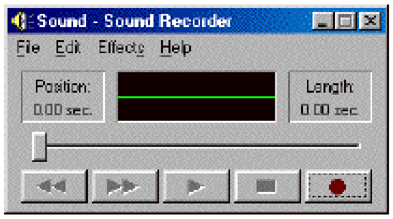
Including audio in your web pages

In our computer lab, you have access to a simple application that comes with Windows 98 called Sound Recorder . While limited, it does provide users with the opportunity to record audio files of up to 60 seconds. It also allows for format conversion. I have tried and experimented with MPEG layer 3 (aka mp3) and liked the quality (although the volume is a bit low) and the file size. Please note that you cannot record in mp3 format! Rather, you can record audio using the default setting and then convert the file format.

Here are the steps:

Open Sound Recorder (Accessories:Entertainment:SoundRecorder)

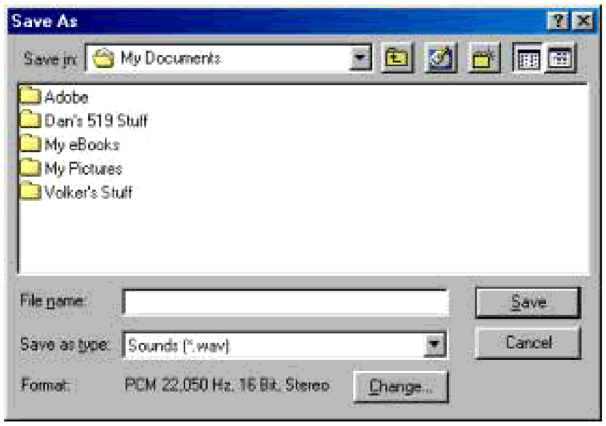
1. Click on the red record button to begin recording using the default settings (Fig. 1)



Position: 0.00 sec.
Length: 0.00 sec.

Figure 1: Default window when opening SoundRecorder

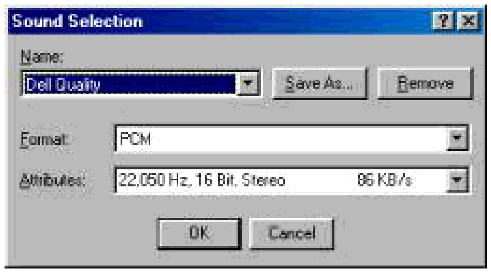
2. Select File:Save. A window like Fig. 2 appears. As you can see the default file type is a wav file. You may save your file as a .wav file or proceed as illustrated below.



Save in: My Documents
File name:
Save as type: Sounds (*.wav)
Format: PCM 22,050 Hz, 16 Bit, Stereo

Fig. 2: Save File dialog window.

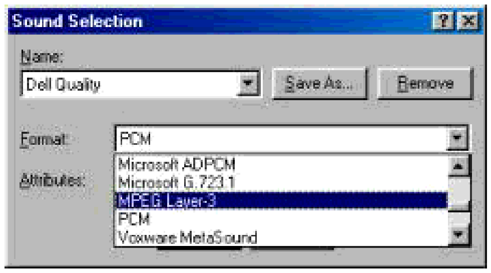
3. Click on Change... to get to Figure 3. Under Format, select MPEG Layer-3 as shown in Fig. 4.


Sound Selection

Name: Dell Quality

Format: PCM

Attributes: 22,050 Hz, 16 Bit, Stereo 86 KB/s


Sound Selection

Name: Dell Quality

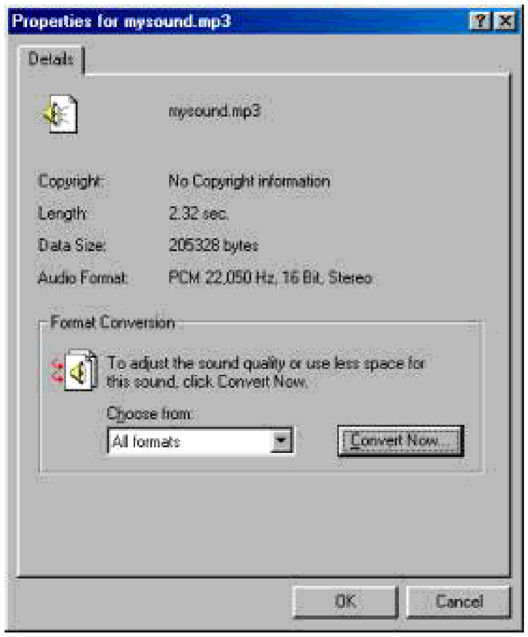
Format: PCM

Microsoft ADPCM
Microsoft G.723.1
MPEG Layer-3
PCM
Voxware MetaSound

Fig. 3: Change Format Window

Fig. 4: Changing the file format to mp3

4. To convert the format of a given sound file, open a file and select File:Properties to get to Figure 5 and click on Convert Now to get to Fig. 3


Properties for mysound.mp3

Details

mysound.mp3

Copyright: No Copyright information
Length: 2.32 sec.
Data Size: 205328 bytes
Audio Format: PCM 22,050 Hz, 16 Bit, Stereo

Format Conversion

To adjust the sound quality or use less space for this sound, click Convert Now.

Choose from: All formats

Figure 5: Changing the file format of an audio clip

The next step is to include or link to the audio files within your html pages.

The following page contains a few examples of embedded audio files (Note: This page contains a number of audio file and will take a little bit to load. If you are accessing this page using a modem connection, be prepared to wait.) This page includes an audio file that starts playing automatically. The user has no way of manipulating it (i.e., it cannot be stopped)

McIntosh

MC-240 Stereo Tube Amplifier.

ELECTRICAL STATS: 40w/ch stereo, 80w mono. Response 16-40kHz (+0 -0.1dB). Distortion 0.5%. Noise & hum -90dB. Output impedance 4, 8, 16, 125 and 600 ohms. 600 ohm center tap internally grounded. (2, 4, 8, 16 and 32 ohms in mono). Output voltages 25 (isolated), 70.7 (one side grounded) and 140V (center tap grounded). Internal impedance less than 10% of rated impedance. Input impedance 250k. Input sensitivity: 0.5V (in mono or twin amp). 2.0V in stereo.

MECHANICAL STATS: Chrome chassis. Line output octal socket for higher impedances and voltages. Barrier strips for 4, 8 and 16 ohms. Audio inputs: stereo, twin and mono. Gain controls: twin amp, balance and mono. Input switch: stereo, twin amp or mono. Can be used for 80w mono. Line voltage switch: 117 or 125V.

TUBES: 4-6L6GC/7027A output, 3-12AX7, 2-12AU7, 2-12BH7 Solid state rectifier.

Size 8"H, 10-3/4"W and 17-1/4"D. Weight 56 lb. Sold from 1960-1969. Last retail price $288.00. Currently these amps sell for around $1000.00 to $1500.00.

Advantages of tube amplifiers:

Much lower distortion at all frequencies, ability to run at full power with no loss, and in the case of McIntosh, many patented technologies. These will be explored in depth as we go on.

Disadvantages of tube amplifiers:

Weight of power transformers needed to drive the high voltage tubes, heat generated by tubes, the fragility of tubes and the size of tubes all made the industry go to transistors for their superior characteristics in these areas. However, this move was never able to reproduce sound as cleanly as tube power.

The McIntosh Autoformer:

The exclusive McIntosh Autoformer is one of the features that make McIntosh Amplifiers unlike any other in performance and reliability. The Autoformer is like one half of a transformer, consisting of one continuous winding with multiple taps for the various output impedances.

Best for your speakers

The output section of receivers; integrated amplifiers and power amplifiers utilize output transistors. When a transistor fails, it often shorts so that it becomes like a solid piece of wire. In a direct-coupled amplifier the output transistors are directly connected to the output terminals. Transistor failure means the power supply of your amplifier could be directly connected to your loudspeakers sending Direct Current (DC) to them. This would be like connecting your loudspeaker to several car batteries connected in series and will severely damage your loudspeakers. Amplifiers that lack Autoformers often use a relay together with the electronic circuitry that disconnects the amplifier output from the loudspeakers. *See figure 1* When a problem occurs, sensing circuitry must first detect the problem, and then tell the relay to disconnect the output from the loudspeakers. This method of speaker protection is not instantaneous, nor is it foolproof. The relay contacts can fuse together before the relay has time to open. McIntosh

Amplifiers with Autoformers are not direct coupled, as the Autoformer is between the output transistors and the loudspeakers. *See figure 2*. One end of the Autoformer Winding in is connected to chassis ground and presents a much lower resistance than your speakers, to the DC that might come from your power supply in the event of a transistor failure. As electricity always takes the path of least resistance, the DC goes through the Autoformer to ground rather than to your speakers. Since the Autoformer is always in the signal path, there is no delay in the protection of your loudspeakers. **The McIntosh Autoformer is the only, always working, 100% foolproof method of preventing Direct Current from reaching your valuable loudspeakers.**

Figure 1: Electro-Mechanical Speaker Protection:

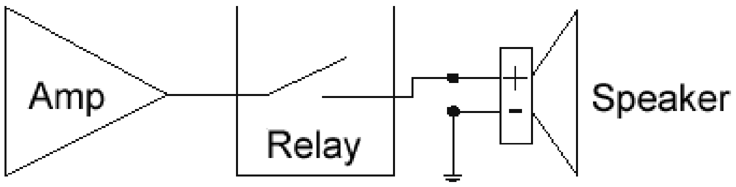
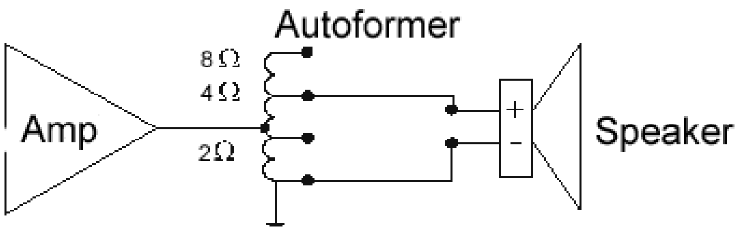


Figure 2: McIntosh Autoformer:



Why do Autoformers solve the problem of Speaker to Amp Impedance miss-match?

Direct Coupled Amplifier's output Circuits are designed with specific loudspeaker impedance. When the output circuitry sees the optimal impedance it delivers full power. If a direct-coupled amplifier is designed for 4 ohms it will only deliver half its rated power into an 8-ohm loudspeaker. See figure 4. This same amplifier, when trying to drive a 2-ohm loudspeaker, will deliver more power, however, it will generate much more heat. See figure 3. Overheated output circuitry will shorten the life of the unit. **The McIntosh Autoformer assures full power will be delivered to all of it's output taps (2, 4 or 8 ohm) and will provide a better impedance match between the output circuitry and your loudspeakers, so your amplifier lasts longer and you always get ALL the power you paid for.**

Figure 3:

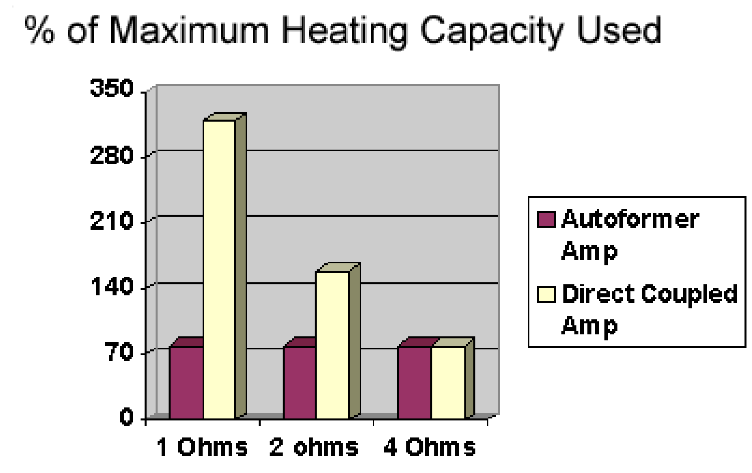
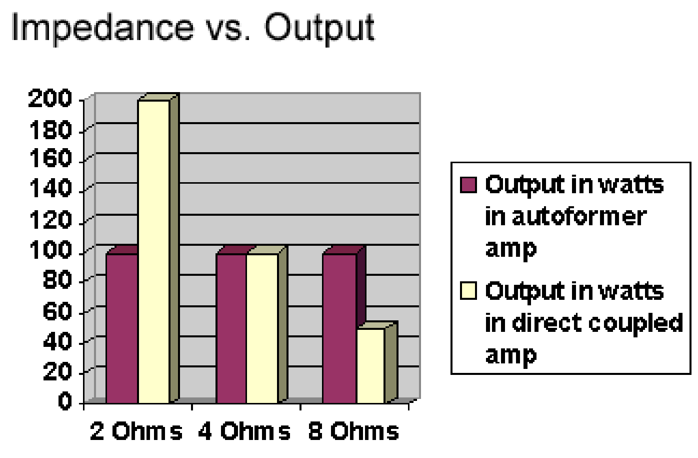


Figure 4:



What is a balanced amp?
Why is a McIntosh balanced amp better than everybody else's balanced amps?

The Fully Balanced Circuit
Some McIntosh amplifiers are designed with fully balanced circuitry all the way from input to output. Each channel in the balanced amplifier circuitry consists of two amplifier sections, one amplifying the positive half of the wave (above the zero line), and the other amplifying the negative half of the wave. Both the positive and negative amplifier sections work into the Autoformer where they are magnetically combined. This forces any distortions inherent in the balanced amplifier circuitry to be produced out of phase from each other and cancel out when the two stages are magnetically combined in the Autoformer. To be fully balanced, an amplifier must have balanced inputs; fully balanced complementary circuitry and Autoformer output. **This fully balanced circuitry provides you with the very low distortion figures only McIntosh amplifiers with Autoformers can provide.**

Figure 5: Bridged Direct Coupled Amplifier:

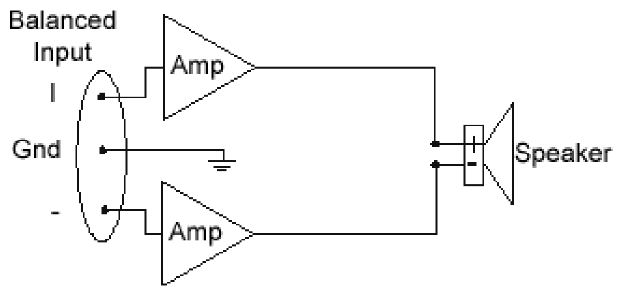


Figure 6: McIntosh Amplifier with Autoformers in Bridged Mode:

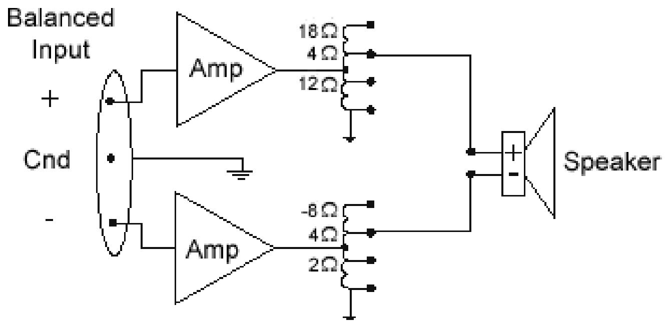
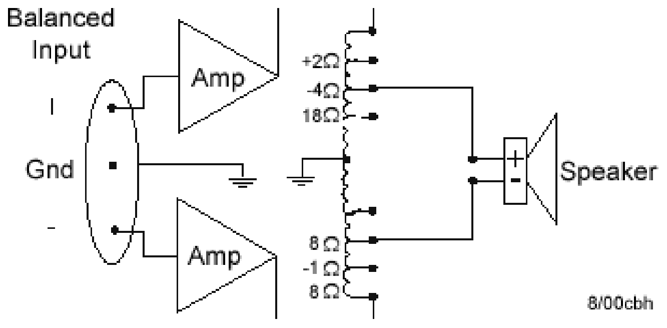
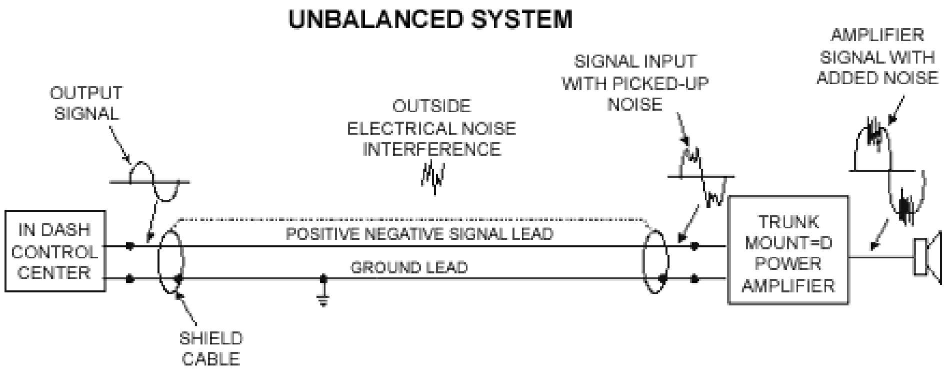


Figure 7: Amplifier with Positive and Negative Waveform Halves Separately Amplified and Combined in an Autoformer:

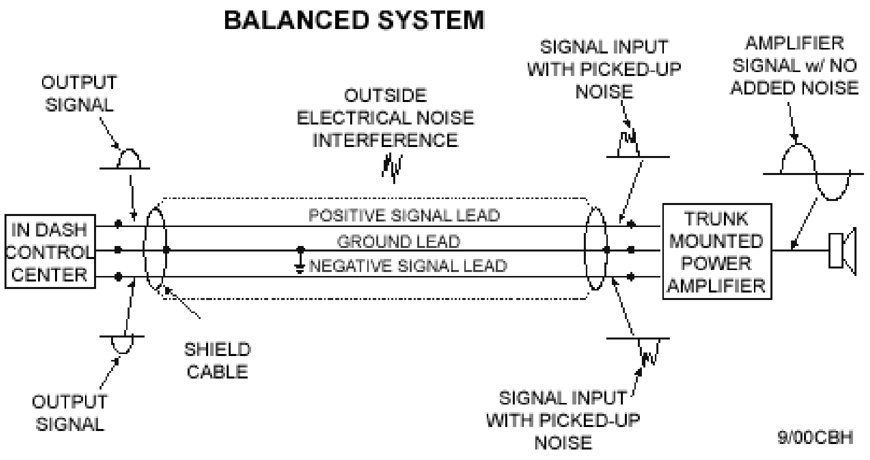


McIntosh balanced inputs:

Any time a signal passes through a cable, outside electrical noise interference can be induced, even when using shielded cables. In an unbalanced system, both the positive and negative halves of the waveform travel together down the Positive and Negative Signal Lead and are influenced by the outside noise to create the Signal with Picked up Noise components illustrated in figure 1.


UNBALANCED SYSTEM
OUTPUT SIGNAL
OUTSIDE ELECTRICAL NOISE INTERFERENCE
SIGNAL INPUT WITH PICKED-UP NOISE
AMPLIFIER SIGNAL WITH ADDED NOISE
IN DASH CONTROL CENTER
POSITIVE NEGATIVE SIGNAL LEAD
GROUND LEAD
TRUNK MOUNT=D POWER AMPLIFIER
SHIELD CABLE

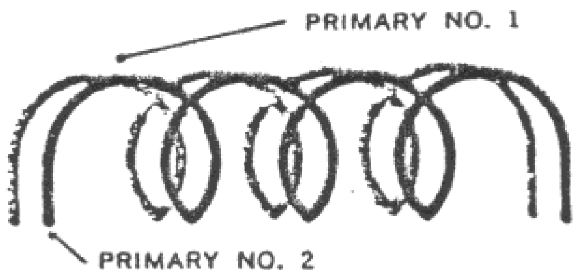
In a balanced system, the positive and negative halves of the waveform are separate; when these separate halves pickup the same outside electrical noise interference, the noise components on one half are out-of-phase with the noise components on the other half (almost like a mirror image). When the negative and positive halves of the signal are combined in the Differential amplifier, the out-of-phase noise components on the two halves cancel each other out as they are combined, leaving only the original output signal. This produces the Amplifier Signal with No Noise illustrated in figure 2.


BALANCED SYSTEM
OUTPUT SIGNAL
OUTSIDE ELECTRICAL NOISE INTERFERENCE
SIGNAL INPUT WITH PICKED-UP NOISE
AMPLIFIER SIGNAL w/ NO ADDED NOISE
IN DASH CONTROL CENTER
POSITIVE SIGNAL LEAD
GROUND LEAD
NEGATIVE SIGNAL LEAD
TRUNK MOUNTED POWER AMPLIFIER
SHIELD CABLE
OUTPUT SIGNAL
SIGNAL INPUT WITH PICKED-UP NOISE
9/00CBH

The McIntosh Unity Coupled Circuit and Bifilar Windings:

The McIntosh Unity Coupled Circuit and Output Transformer have established McIntosh amplifiers as the unchallenged leaders in the audio field. Before 1947, low distortion at high power and high efficiency was impossible. A completely new engineering approach resulted in an amplifier that for the first time permitted high power with distortion below 1%. That new engineering produced the McIntosh Unity Coupled Circuit and the McIntosh Bifilar Wound Output Transformer. With the introduction of the McIntosh amplifier new standards for distortion free performance were established. The McIntosh output transformer is unique. It has two primary windings, which are wound bifilarly. In the bifilar technique both primary wires are wound side by side. Each turn of primary number one is next to the same turn of primary number two. There is almost complete magnetic coupling between the two wires. The magnetic coupling is reinforced by the capacitance between the two wires.

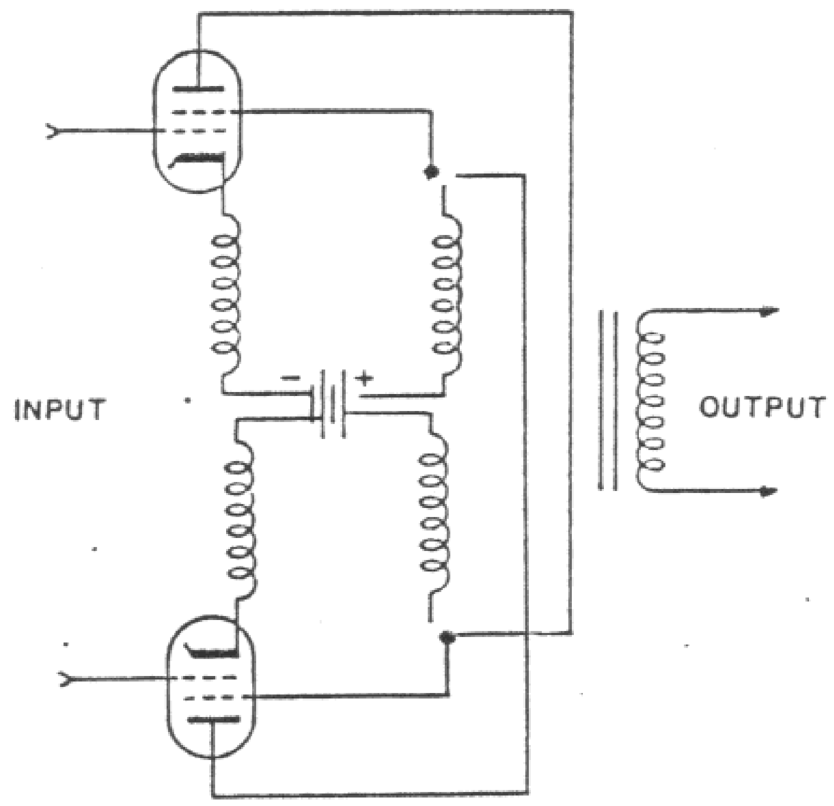
Example of Bifilar winding



In the McIntosh Unity Coupled circuit one of the bifilar primary windings is connected through the power supply to the plate and cathode of one of the output tubes. The other bifilar primary winding is similarly connected to the other tube. All low distortion high power amplifiers use push-pull output circuits known as class AB1, AB2 or B. Two tubes are arranged in a balanced circuit. This permits each tube to operate alternately somewhat over half the time. Compared to full time operation of the tubes, the push-pull method reduces heating and permits more power from a given type of tube. Despite this advantage of the conventional push-pull circuit one problem in particular remained to be solved. When current in each tube is cut off to begin the idle period distortion is produced at the instant of cut off and again at the instant when current flows. This form of distortion is known as Notch Distortion. Imperfect couplings between the primary windings found in all conventional output transformers produce the condition which permits notch distortion. Trying to improve coupling in a conventional transformer decreases the power response at both low and high frequencies, heating the output tubes and lowering the available power output. The McIntosh Unity Coupled output circuit and bifilar transformer is the first commercial breakthrough that eliminates notch distortion by coupling both output tubes almost to perfection. In the McIntosh transformer, the extremely close coupling of the bifilar windings removes the condition that permits notch distortion. Furthermore, the two output tubes are arranged as partial cathode followers. Half of the output circuit is in the cathode and half in the plate of each tube. The output tubes now are operating in a local feedback loop, reduces their distortion, reduced their internal generator resistance, and reduces their balance requirements. The McIntosh circuit, in reality, perfects push-pull high efficiency output circuits.

Leakage inductance (lack of coupling) between the primary and secondary windings of the output transformer limits the high frequency response of an amplifier. The primary and secondary windings of the McIntosh output transformer are interleaved five times to improve coupling. The interleaving is accomplished by winding groups of primary layers, then secondary layers alternately until the total transformer is wound. Interleaving helps to extend the McIntosh power bandwidth to over 100,000 cycles.

McIntosh Output Circuit

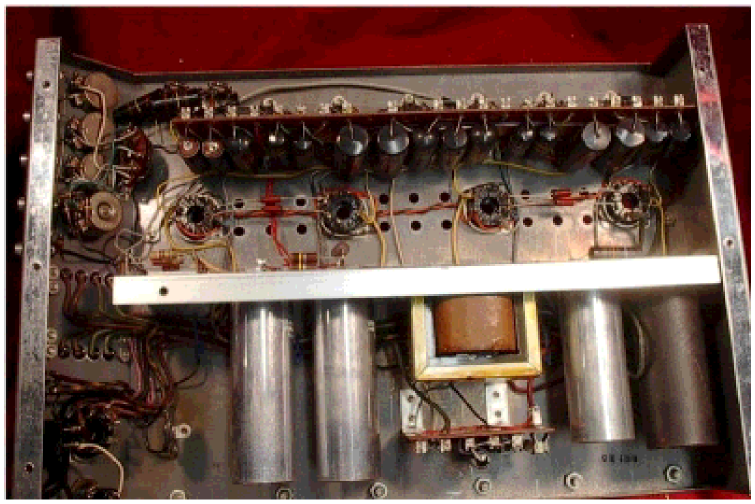


Other features of tube McIntosh gear:

Good voltage regulation in the power supply permits overloads without overshoot or blocking, good transient response, and complete stability. To improve regulation a silicon rectifier power supply is used in the MC240. In addition to better voltage regulation, the silicon rectifier allows even higher operating efficiency, cooler operation, and longer amplifier life.

To greatly extend tube and component life a thermistor in the MC240 limits current surges produced when the equipment is turned on. The thermistor is a special type of resistor. Its resistance depends on its temperature. When the amplifier is off the thermistor has a high resistance value (about 79 ohms). Just after the amplifier is turned on, the current that flows through the thermistor heats it and causes its resistance to decrease to a low value (less than .7 ohms). Current is thus limited when the MC420 is first turned on but is not limited as the unit warms.

Internal view of the McIntosh 240



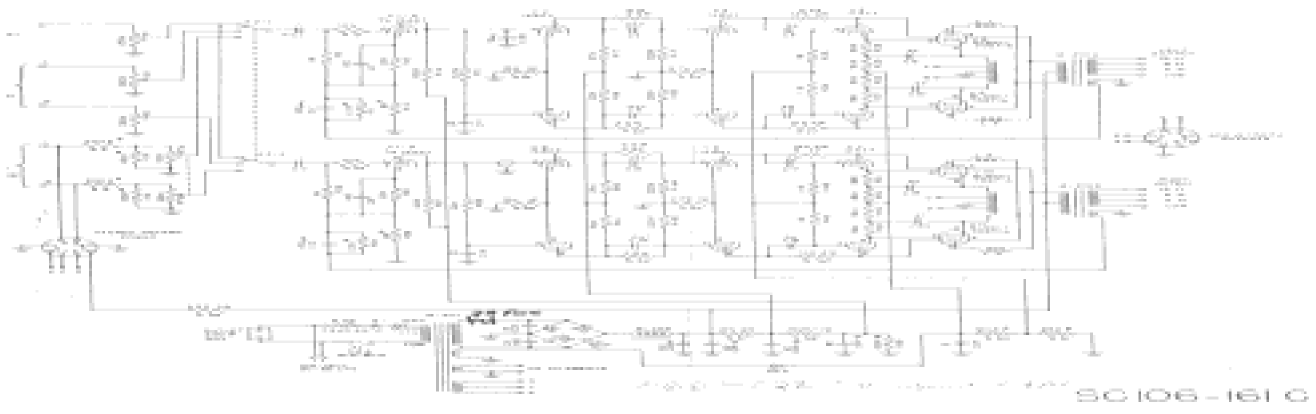
A new model McIntosh Tube Amplifier

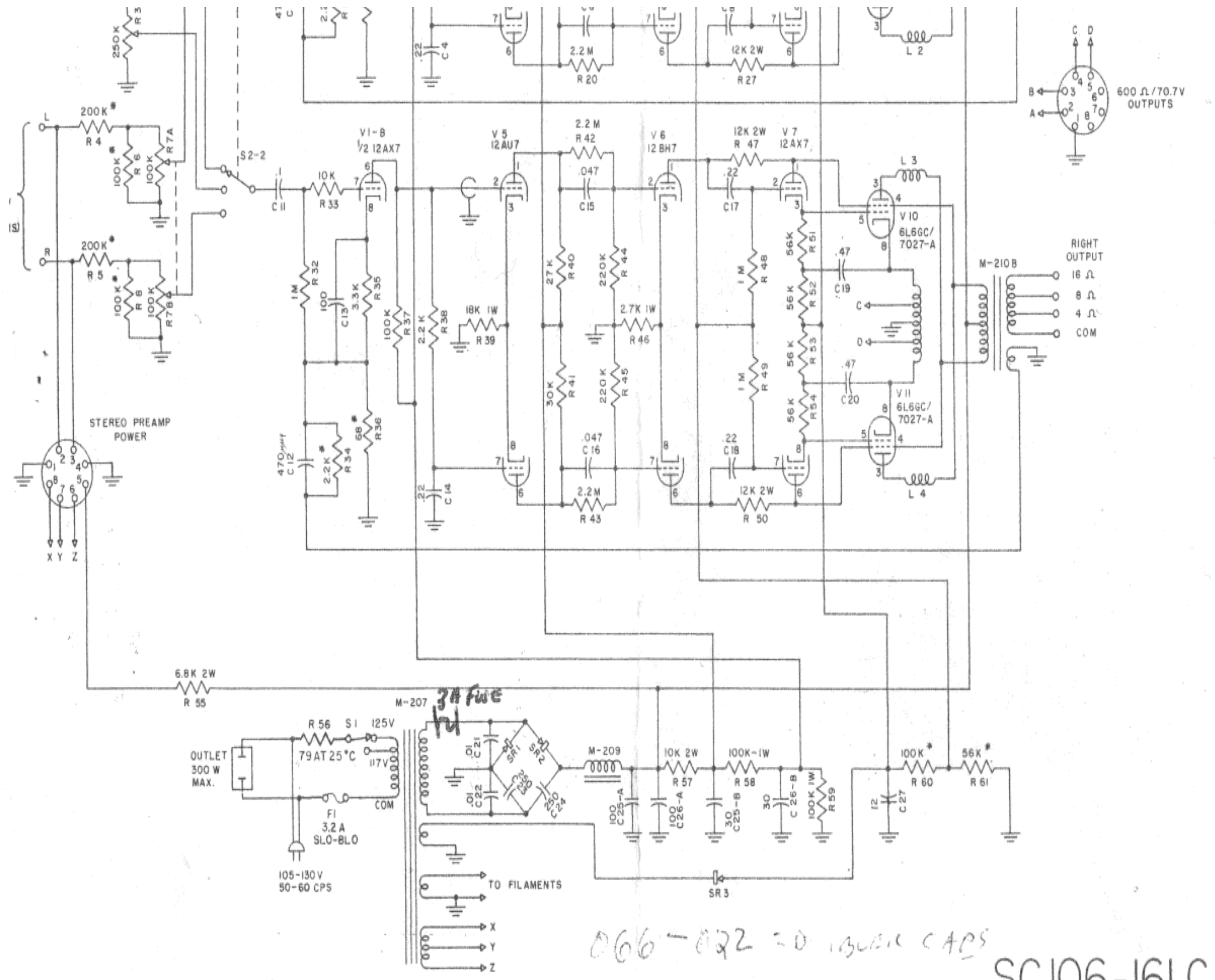
Links

McIntosh Labs HomePage

AutoSwitcher for Amplifier testing

Schematics for the MC240





066-022 20 [illegible] CAPS

SC106-161C

VOLTAGE AND RESISTANCE CHART

Tube	Pin No.	DC Volts No. Signal	AC Volts at rated output	Resistance in Ohms Unit Off
12AX7 (V1)	[illegible]	[illegible]	[illegible]	[illegible]
12AU7 (V2 or V5)	[illegible]	[illegible]	[illegible]	[illegible]
12BH7 (V3 or V6)	[illegible]	[illegible]	[illegible]	[illegible]
12AX7 (V4 or V7)	[illegible]	[illegible]	[illegible]	[illegible]
6L6GC/7027A (V8, V9, V10, V11)	[illegible]	[illegible]	[illegible]	[illegible]

*This Resistance measured with condenser C25A shorted to ground.
A high impedance VTVM is used to measure the operating voltages and resistances.
BE SURE THE AMPLIFIER IS TURNED OFF WHEN MEASURING RESISTANCES.

12AX7	1	103	1.9	160K*
(VI)	2	0	.42	1M
	3	.7	.41	3.3K
	4 & 5	0	0	0
	6	103	1.9	160K*
	7	0	.42	1M
	8	.7	.41	3.3K
	9	0	6.3	0
12AU7	1	205	11	40K*
(V2 or V5)	2	103	1.9	160K*
	3	108	.9	18K
	4 & 5	0	0	0
	6	205	11	43K*
	7	85	0	2.4M*
	8	108	.9	18K
	9	0	6.3	0
12BH7	1	350	144	12K*
(V3 or V6)	2	18	11	200K
	3	35	.6	2.7K
	4 & 5	0	0	0
	6	350	144	12K*
	7	18	11	200K
	8	35	.6	2.7K
	9	0	6.3	0
12AX7	1	430	108	26*
(V4 or V7)	2	−46	144	1.1M
	3	−46	136	260K
	4 & 5	0	0	0
	6	430	108	26*
	7	−46	144	1.1M
	8	−46	136	260K
	9	0	6.3	0
6L6GC/7027A	1	—	—	—
(V8, V9, V10, V11)	2	0	0	0
	3	430	108	27*
	4	430	108	26*
	5	−46	136	260K
	6	—	—	—
	7	0	6.3	0
	8	.7	108	26

*This Resistance measured with condenser C25A shorted to ground.

A high impedance VTVM is used to measure the operating voltages and resistances.

BE SURE THE AMPLIFIER IS TURNED OFF WHEN MEASURING RESISTANCES.

PIN 2 — L INPUT
PIN 3 — R INPUT
PIN 4 — GROUND
PIN 5 — 375 V DC AT 15MA
PIN 6 — 25.2 V AT 1.2 A
PIN 7 — 25.2 V CENTER TAP
PIN 8 — 25.2 V AT 1.2 A

600 Ω / 70.7 V OUTPUT RECEPTICAL CONNECTIONS

PIN 1 — GROUND, 70.7 V RETURN, AND 600 Ω C.T.
PINS 1 AND 2 — LEFT 70.7 V OUTPUT
PINS 2 AND 3 — LEFT 600 Ω OUTPUT
PINS 1 AND 4 — RIGHT 70.7 V OUTPUT
PINS 4 AND 5 — RIGHT 600 Ω OUTPUT

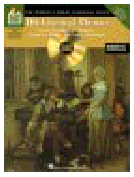
SINGLE CHANNEL (MONO) OUTPUT NOTE:

FOR SINGLE CHANNEL OUTPUT
PARALLEL OR SERIES CONNECT THE
L AND R OUTPUTS TO OBTAIN
DESIRED OUTPUT IMPEDANCE.

✳ 5% TOLERANCE RESISTORS

FOLLOWING COMPONENTS ARE
MATCHED TO 1%:
R 11 AND R 34
R 13 AND R 36
R 24 AND R 27
R 47 AND R 50

Convert MIDI to WAVE Files



Directions: how to put MIDI files on CD

Before I begin to explain this, understand that MIDI files don’t contain any **sounds** so we need to use some sort of synthesiser-- either an external one, or the one in your computer and record this synth as a wave file.

The first question you must answer is: Are you happy with the quality of the sounds of these MIDI files right now? Do you want to record them as they sound right now in your computer using your computer’s synth? If so, then it’s simple. Just read on. If not, scroll to the bottom and read the directions below.

You need to open your volume controls (double click the yellow speaker icon in the tray (the tray is the clock area of your Windows computer)).

Check the ’Mute’ check boxes for all sliders except ’Synthesizer Balance.’

Now select ’Options’ and ’Properties.’ Click ’Recording’. Press OK.

Along the bottom of this "Recording Control" will be Select check boxes. Select ’Mixed Output.’

Now your system is ready to record the synthesizer as a wave file. You need to open up two programs next: the wave file recording program (I recommend CoolEdit2000) and the MIDI
program you use for playing MIDIs.

You can adjust the sample rate of the recording using CoolEdit. For other wave packages you’ll have to scan the help files to figure out how high to set the sample rate.

It takes longer, but for the best quality, record at a high 48000 sample rate, stereo, and 16 bit. The CD won’t allow you to record at this high, so then resample the recording to 44100 quality. You’ll get the best sound on your CD this way.

Start recording using CoolEdit, and while it’s recording, switch to the MIDI program and press ’play.’

When the song is over, press ’stop’ in both programs: CoolEdit and the MIDI program. Save the new wave file in CoolEdit as a PCM windows wave file. Before you save you might want to delete the beginning silence and the end silence of the wave file. That’s up to you.

Now try playing the WAVE file to ensure it sounds as good as you hoped. If not, it could be that your recording settings need to be changed (maybe the volume on ’Synth’ was too high). Some sound cards have a noisy record feature so it might be that.

If you do NOT want the quality of sound that comes from your sound card, then I assume you’ll be using an external synthesizer for sounds.

If this is the case, you go through the same steps as above, but you’ll want to send the external synthesizer sounds INTO your sound card. The way to do this is to get a Mini plug that will go from your synthesizer out, or headphone out into the Line In plug on the back of your computer. When you are muting sliders in the Volume Control, mute everything except ’Line Balance.’

If you don’t have a synthesizer, then you’ll have to get one if you’re not happy with the sounds in your computer.

I hope this helps you. I assume you know how to get the wave files onto a CD. If not, let me know.

MP3 101

So, how does MP3 technology work? Well, to put it simply, it takes music and takes out the unnecessary parts and then uses mathematics to compress it. But that's the bare bones basics. The idea behind this page is to show you a little more in detail how this technology actually works.

Your average song takes up about 60Mb of space. However, a MP3 song can take between 3-10Mb, depending on the recording quality. That's nearly 10 times smaller then the original. That's is why MP3 are such a big deal in the music industry.

The existing JPEG (Joint Photographic Experts Group) standard for bit rate reduction had been developed for still pictures, and was clearly inadequate for moving pictures. To avoid a crippling standards battle, the Moving Pictures Expert Group (MPEG) was formed to devise a suitable coding scheme for transmitting moving pictures and recording them in standard digital storage media, i.e. CD-ROM, CD-i, Multi-session CD, Video CD.

A meeting between the International Standards Organisation (ISO) and the International Electrotechnical Commission (IEC) in 1992 resulted in a standard for audio and video coding, known as MPEG1 (ISO/IEC 11172).

MPEG2 (ISO/IEC 13818) became a standard on 11th November 1994 after a five day meeting of ISO and ITC in Singapore.

Today MPEG2 delivers picture and sound quality equal to TV studio standards. The next step was MPEG3 which is the format widely used today for MP3 music.

How does MPEG audio work?

In devising an audio coding method, the basis had to be the human ear. Unfortunately this is not a perfect device for acoustic reception but is the best that

we have. Advantage was taken of one of the human ear’s shortcomings: its non-linearity and adaptive threshold of hearing.

The threshold of hearing is the level below which a sound is not heard. It varies with frequency and, of course, between individuals. Most people’s hearing is most sensitive between 2 and 5 kH. Whether a person hears a sound or not depends on the frequency of the sound and whether its amplitude is above or below that persons hearing threshold at that frequency.

The threshold of hearing is also adaptive, being constantly changed by the sounds heard. For example, an ordinary conversation in a room is perfectly audible under normal conditions. However, the same conversation in the vicinity of a loud noise, such as an aircraft passing low overhead, is impossible to hear due to the distortions introduced to the hearing thresholds of the individuals concerned. When the aircraft has gone the hearing thresholds return to normal. Sounds that are inaudible due to dynamic adaptation of the hearing threshold are said to be "masked".

This effect is universal but is of particular relevance in music. An orchestra instrument playing fortissimo will, to a greater or lesser extent, make the sound of some other instruments inaudible to the human ear. When the music is recorded, however, all the frequencies go on the medium because the response of the recording device is flat, i.e. it is not dynamically adaptive. When the recording is played the masked instruments will not be audible to the listener, so might as well not be there. A linear recording, as used on CD, is inefficient in this respect. To make the most use of a recording medium the parts of the medium that contain inaudible data can better be used for audible data. In this way the amount of the recording medium needed to contain the music can be considerably reduced without any loss of audio quality.

Tutorial

Ultrasound is very useful for imaging structures in the body. Ultrasound is a nonionizing form of energy that propagates through a medium as pressure waves. If you could measure the very small pressure disturbance when an ultrasound wave travels by, you would find it fluctuates rapidly about the ambient, normal background pressure until the wave is past.

One way we characterize a sound wave is by its frequency, that is, the number of oscillations or fluctuations per second in the medium. Audible sounds have frequencies between about 15 cycles per second (15 hertz, Hz) and 20,000 cycles per second (20 kHz). The upper limit of human hearing is usually taken to be 20 kHz, and ultrasound refers to sound waves whose frequency is above this level. Other mammals are not nearly as limited as man in terms of the useful sonic frequency range. For examples, bats and dolphins utilize ultrasound waves that have frequencies as high as 125 kHz for navigating and sonar visualization.

Medical ultrasound applications have been described that use frequencies from as low as 500 kHz to as high as 30 MHz. For most imaging applications, ultrasound devices operate between 3.5 MHz and about 10 MHz. The exception is for intravascular imagers, tiny catheter tipped probes operating at frequencies as high as 30 MHz. The optimal ultrasound frequency for any application represesents a tradeoff between a) the need to acquire ultrasound images with a high degree of spatial resolution, dictating use of higher frequencies, and b) the need to obtain adequate "penetration" in the tissue. Imaging depth into tissue is limited by attenuation of the ultrasound waves, and this becomes more severe as the ultrasound frequency is increased.

The most commonly used modality in medical ultrasound is called B-mode imaging. An ultrasound transducer is placed against the patients skin surface, directly over the region to be imaged. The transducer sends a very brief pulse of ultrasound into the tissue. The pulse travels along a beam, very much like the beam of a search light at an airport. Interfaces along the way reflect some of the ultrasound energy back to the transducer. The transducer, in turn, converts the reflected energy into echo signals, which are sent into amplifiers and signal processing circuits inside the imaging machine’s hardware. The exact, microsecond delay between when the transducer first launched the ultrasound pulse and when it picked up an echo tells the machine how far the reflecting interface is from the transducer.

After all the echoes are picked up from along the first beam, the transducer sends a second pulse along a slightly different beam direction into the tissue. Echoes are picked up the same way and sent into the machine hardware.   Then another pulse is launched in still a different direction, and so forth. Very much like the beam of a search light is swept across the night sky, the pulsed beam from an ultrasound transducer is swept throughout the body, mapping out reflectors and other interfaces and forming 2-dimensional images.

Tutorial 2 – Ultrasound

1.
(a) Sketch the major components in an ultrasound transducer array and discuss their main characteristics.
(b) Show how a linear array may be used to both focus and direct an ultrasound beam, extend this to a multi-dimensional array

Q(a) and (b) are book work see lecture notes.

(c) In a B-mode ultrasound scanner, the frequency of the transducer is 7.0 MHz. The scanner has a frame rate of 50 frames/s and the linear transducer array consists of 256 elements with a 64 element subaperature that are fired simultaneously to form one line scan. Calculate the depth of penetration. Suggest an application for this transducer. What could you do to reduce the depth of penetration?

Answer:

$$F \times N \times D = \frac{c}{2}$$

$$so\ D = \frac{1540}{2 \times 50 \times (256 - 64 + 1)} \approx 8cm$$ *and therefore this system is suitable for relatively superficial imaging (eg surface muscles). To reduce the depth of penetration we could increase the frame rate or decrease the size of the subaperature eg make it say 32 elements.*

2. To avoid ambiguity in pulsed Doppler and pulse echo imaging, the echoes from the deepest structures in the body must be received before the next pulse is transmitted. Consequently, the depth of penetration Zmax, determines the PRF of a pulsed Doppler or echo imaging system. Given that the PRF has to be at least twice the as large as the maximum Doppler frequency, show that $Z\max V\max \le \frac{c^2}{8f}$ where Vmax is the maximum velocity, c is the ultrasound velocity in blood, and f is the ultrasound frequency. Assume normal incidence.

Answer:

$$\frac{1}{PRF} = \frac{2Z_{\max}}{c}$$
$$f_{d\max} = \frac{2v_{\max}\cos\theta f}{c}$$
$$\therefore \frac{c}{2Z_{\max}} = \frac{4v_{\max}\cos\theta f}{c}$$
$$v_{\max}Z_{\max} = \frac{c^2}{8f}$$

3. Calculate the scattering cross-section at 2MHz of an air bubble used as a contrast agent in ultrasound, where the bubble is 3 microns in radius and the compressibility and density of air are 6.9 X 10^{-7} cm^2/dyn and 1.3 X 10^{-3} g/cm^3, respectively; while those for water are 4.6 X 10^{-11} cm^2/dyn and 1.02 g/cm^3. Assume the speed of sound in water is 1480 m/s. How does this compare to the scattering from a red blood cell.

$$\sigma_s = \frac{4\pi k^4 a^6}{9}\left[\left|\frac{G_s - G}{G}\right|^2 + \frac{1}{3}\left|\frac{3\rho_s - 3\rho}{2\rho_s + \rho}\right|^2\right]$$

$$\lambda = 1480/2\times10^6 = 740\times10^{-6}$$

$$\sigma_s = \frac{4\pi\left(\frac{2\pi}{740\times10^{-6}}\right)^4\left(3\times10^{-6}\right)^6}{9}\left[\left|\frac{6.9\times10^{-7} - 4.6\times10^{-11}}{4.6\times10^{-11}}\right|^2 + \frac{1}{3}\left|\frac{3\times1.3\times10^{-3} - 3\times1.02}{2\times1.3\times10^{-3} + 1.02}\right|^2\right]$$
$$= 1.19\times10^{-9} m^2$$

this is much greater than the scattering cross-section of a red blood cell.

Beams are swept and ultrasound images are formed very rapidly, essentially in "real-time. As the operator holds the transducer in contact with the skin, the image appears live on a video monitor. By moving and manipulating the transducer different internal views are provided. The images represent cross-sectional, or tomographic views of the plane that the beam was swept across in.

Ultrasound provides images of any region in the body where there is a soft tissue path between the probe - or ultrasound transducer - and the region of interest. In abdominal imaging, for example, with the transducer placed on the skin surface beneath the rib cage the sonographer can view the liver, blood vessels inside the liver, aorta, kidneys, pancreas and spleen. With special oral preps, the stomach and even the intestines can be viewed. Diffuse disease conditions, such as cirrhosis and fatty infiltrated liver, focal disease, such as cancerous tumors and vessel abnormalities are detected. Other common ultrasound exampination areas include the heart, the pelvis, the neck and the arms and legs. In some applications it is advantageous to utilize intra cavitary transducers for close up views of the uterus, the ovaries, the prostate and the colon.

More to come...

The GIF File

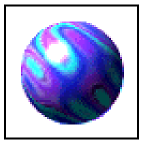
First, let’s look at the characteristics of a GIF file. GIF stands for "Graphic Image Format", and the format was developed by Compuserve to provide a means of passing an image from one dial-up customer to another, even across different computer hardware platforms. It is a relatively old fomrat, and was designed to handle a palette of 256 colors (8 bit color), and a single image. When developed, this was near state of the art for most personal computers.

The "GIF" format uses an 8 bit Color Look Up Table (sometimes called a CLUT) to identify its color values. If the original image is an 8 bit, gray-scale photo, then the "GIF" format produces a compressed *lossless* image file. (*Lossless* means that the image uncompressed from the file would be identical to the original.) A grey scale image typically has only 256 levels of gray. This is accomplished by the Run Length Encoding mechanism of compressing the information while saving a GIF file.

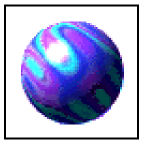
If the orignal file were a 24 bit color graphic image, then it would first be mapped to an 8 bit CLUT, and then compressed using RLE (Run Length Encoding). The loss would be in the remapping of the original 24 bit (16.7 million) colors to the limited 8 bit (256 colors) CLUT. RLE encoding would reproduce an uncompressed image that was identical to the remapped 8 bit image, but not the same as the original 24 bit image.

RLE is not an efficient way of compressing an image when there are many changes in the coloration across a line of pixels. It is very efficient when there are rows of pixels with the same color or when a very limited number of colors is used. A controllable parameter in SAVING GIF files is the pallette size. In many software packages the bit depth of the palette can be specified before an image is saved. This is sometimes called color "indexing". The follwing image has been indexed to each of the possible values.

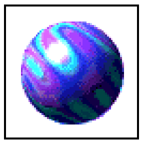
8 bit = 256 colors



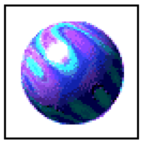
7 bit = 128 colors



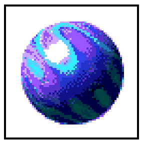
6 bit = 64 colors



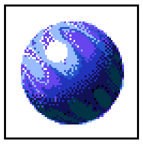
5 bit = 32 colors



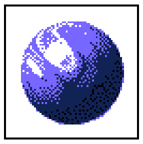
4 bit = 16 colors



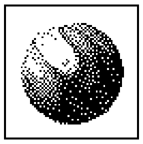
3 bit = 8 colors



2 bit = 4 colors

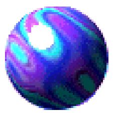


1 bit = 2 colors (usually B&W)



As you can see the quality of the original image decreases as the bit depth decreases. Because the RLE compression is lossless, a GIF file can be repeatedly saved after modification without any additional quality loss, as long as the color depth remains unchanged.

The GIF format provides one unique quality that is very useful to WEB authors. One color in a lookup table can be identified as TRANSPARENT. Any pixels that are mapped to the TRANSPARENT color will pass the background color of the page through. This is often used to make circles, arrows and other irregular shaped objects lose their rectangular frames.

GIF File without transparency		GIF File without transparency	
GIF File with transparent exterior white color was mapped to transparent		GIF File with transparent exterior white color was mapped to transparent	

| WEB INFO | GIF vs. JPEG

The Online POV-Ray Tutorial

Advanced POV-Ray Features

(Show Jump Points) (Hide Jump Points)

If you've made it this far, you're in good shape! This section covers the features of POV-Ray that are most most complex, but also the most powerful. Once you complete this section, you'll be ready a certified ray-tracing master.

Quick Index:

#declare

Up until now, creating large numbers of similar objects has been an excersize in cut-and-paste editor features. POV-Ray provides a very powerful, very flexible way to create many similar objects with a statement called *#declare*. `#declare` essentially creates a new type of object (or pigment, or texture, or almost anything) that you can use and re-use whenever you like. Take a look at the following code:

```
#declare my_sphere =
sphere {
  <0, 0, 0>, 5
  finish {
    pigment rgbf <.5, .2, .4, .667>
  }
}
```

What this does, essentially, is declare a new type of object called "my_sphere" which you can now use later on in your source code, like this:

```
object {
  my_sphere
  translate <-2, 0, 0>
}
```

The *object* statement tells POV-Ray to create an object of type "my_sphere." Theoretically, you can put `object` statements around *every* object you use (including primitives, like spheres) but POV-Ray only requires it for `#declared` objects.

Note that any attributes you place inside the `object` statement override those in the `#declare` statement -- in this example, our sphere is moved from its original location at `<0,0,0>` to `<-2,0,0>`. This hold true for pigments, finishes, etc.

VRML programmers should take note that this `#declare` differs somewhat from VRML's `DFN` node. `#declare` does not create an instance of the object (which `DFN` does), only the definition. In other words, the above `#declare` statement would not add any objects to your scene on its own. You need to instantiate the objects (with the `object` keyword) to do that.

Now, why would you want to use `#declare`? Say, for example, you're making a Greek temple. You would want many pillars in your object, so you would create a pillar object with `#declare`, like this:

```
#declare pillar =
cylinder {
  <0, -5, 0>, <0, 5, 0>
  texture { White_Marble }
}
```

Then, you would create however many of these you needed, translating to your heart's content. Say, however, that you decide the columns in your temple should be made out of *red* marble, not white. All you have to do is change the one `#declare` statement, and all the pillars change! If you had created those pillars without `#declare`, you'd have to change each one by hande -- a major hassle, especially if you had 40 pillars in your temple.

So you can see one immediate benefit to `#declare` -- updating your scene becomes a lot easier. But wait, there's more! You can also use `#declare` to create your own colors and textures. In fact, the `colors.inc` and `textures.inc` files are basically long lists of `#declare`d colors and textures, respectively. The syntax is intuitive:

```
#declare blue_green_filter = rgbf <0, .5, .5, .5>

#declare red_glass =
texture {
  finish {
    refraction 1.0
    reflection 0.1
    ior 1.5
  }

  pigment {
    color rgbf <1, .7, .7, .7>
  }
}
```

As you can most likely guess, these define a new color, called "`blue_green_filter`" and a new texture, called "`red_glass`". You would use these like this:

```
sphere {
  <0, 0, 0>, 1
  pigment { blue_green_filter }
}

cone {
  <-2, -4, 16>, 5
  <0, -3, 1>, 1
  texture { red_glass }
}
```

Not too difficult! You can use `#declare` to create custom `finish`, normal, and `pigment` statements... you can even use it with vectors and single numbers, like this:

```
#declare PI = 3.1415926545338327950288
```

This will save you a bit of typing if you reference PI frequently in your scene file! (Please remember that you don't need to put an `object` statement around anything you `#declare` other than objects).

C and C++ programmers should not be mislead by `#delare`'s superficial similarity to the C/C++ pre-processor macro `#define`. Their behaviour is quite different. `#define` actually changes the source code before it gets compiled (why is why it's called a *pre*-processor macro). POV-Ray does not have a pre-processor, and so `#declare`, although misleadingly labeled, will not do source-code substitution.

At any rate, you can get the complete syntax for object and #declare in the Language Reference. They are both powerful tools, and if you create anything other than very simple scenes, you will find them invaluable.

CSG

CSG stands for **C**onstructive **S**olid **G**eometry, a powerful technique in POV-Ray for creating new objects from combinations of other objects. So far, you have been limited to POV-Ray's primitives, which, while nice, aren't always what you need. POV-Ray lets you use the primitives in a much more constructive (har har) way with CSG: you can carve away parts of objects, you can stick objects together, and other exciting stuff.

There are five operators in CSG: union, intersection, merge, difference, and inverse. The syntax of all the operators (except `inverse`) is very simple: it's the operator, followed by a list of two or more objects enclosed by braces, like this:

```
CSG_operator {
  object_1
  object_2
  etc.
}
```

You actually don't have to put any objects at all between the braces, but it doesn't make sense to have less than two objects (remember, CSG creates new objects from *combinations* of other objects) and POV-Ray will warn you when you trace the file. The syntax for `inverse` is even easier: it's just the word "inverse."

We'll go over these operators one by one, because they're all important. A complete reference can be found in the CSG Section of the Language Reference.

Union

A union is the easiest CSG operator to understand. It simply takes a bunch of objects, and "sticks them together." It doesn't actually *move* the objects at all, but it creates a common bond between the objects, kind of like they've joined a special club for important primitives. (We'll politely ignore the similarities to certain political parties). The source code to a sample union looks like this:

```
union {
  sphere { <0, 1, 2>, 3 }
  box { <-77, 6, 5>, <2, 3, 55> }
  sphere { <-2, -3, -4>, 5 }
}
```

Now rendering the scene doesn't look any different whether you have the `union` keyword there or not. So why bother? Two reasons: first, you can assign attributes to the entire union of objects very easily:

```
union {
  sphere { <0, 1, 2>, 3 }
  box { <-77, 6, 5>, <2, 3, 55> }
  sphere { <-2, -3, -4>, 5 }

  pigment { color Blue } // applies to the entire union
}
```

In this case, the attribute `pigment { color Blue }` is applied to *every* object in the union. As always, this works with any attribute you care to try: pigment, translations, normal, etc.

The second, and perhaps even more useful reason for using unions, is when you combine CSG and the `#declare` keyword, like this:

```
#define two_spheres =
union {
  sphere { <0, 0, 0>, 3 }
  sphere { <-1, -5, -1>, 3 }
}
```

From now on, you can reference the object `two_spheres` (which is, amazingly enough, two separate spheres) just as you would any other `#declared` object:

```
object {
  two_spheres
  pigment { color Pink }
  rotate <0, 180, 0>
}
```

Let's go through one more example, to make sure you understand -- this is a very important concept. Say you wanted to ray-trace a car. You'd create the wheels, then an axle, and then use `union` to stick them together. You could then re-use this wheel and axle combination however many times you wanted (depending on how many sets of wheels your car has). Your code might look something like this:

```
#declare wheels_n_axle =
union {
  object {  // left wheel
    wheel   // assuming we have already created a wheel object
    translate <-3, 0, 0>
  }

  object {  // axle
    axle    // assuming we have already created an axle object
  }

  object {  // right wheel
    wheel   // assuming we have already created a wheel object
    translate <3, 0, 0>
  }
}

#declare car =
union {
  object { // front wheels and axle
    wheel_n_axle
    translate <0, 0, 5>
  }

  object {  // rear wheels and axle
    wheels_n_axle
    translate <0, 0, -5>
  }

  // other car parts go here
}
```

Note that the order you place objects in a `union` is unimportant -- objects within a union don't really care about the other objects. This is different from the objects in a `difference` -- they are very caring, almost loving, objects, as you will see in the next section.

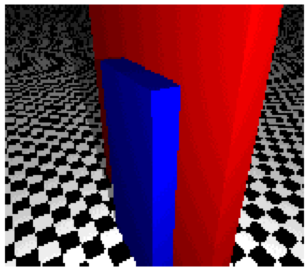
A complete description of the `union` operator can be found in the CSG Section of the Language Reference.

Difference

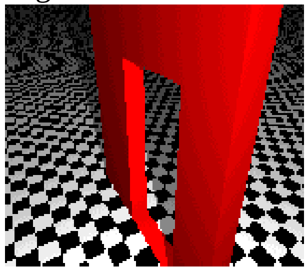
A CSG *difference* is much like a mathematical difference -- it subtracts objects from one another. More specifically, it takes chunks out of the first object, each chunk being defined by the other objects in the `difference` statement. For example, say we wanted to make a wall that we would add a door to. The simplest way to do this is with a `difference`:

```
#declare wall =
difference {
  box { <0, 0, 0>, <10, 10, 1> } // 10x10x1 wall
  box { <2, 0, -1>, <6, 8, 2> } // minus a doorway
  texture { Wall_Texture } // assuming we have already created a Wall_Texture
}
```

The first cube serves as the wall, and the second cube describes what, exactly, we want to take out from the wall. The two objects *without* the `difference` statement look like this:



When we add the difference statement, we get:



Note that we made the doorway cube thicker than the wall. Why? This is because, occasionally, POV-Ray will get confused when you have two objects that overlay exactly the same space. So, we made the doorway cube a little thicker, avoiding a potentially weird image, and at no loss to anything else.

One important thing to remember about differences is that all objects are subtracted from the first one. If, for example, we wanted to add a few window holes to the wall above, we could just add a few more cubes at the very end, and voila! Once again, any attributes placed at the end of the difference statement will apply to the entire object.

A complete reference for the `difference` keyword is located in the CSG Section of the Language Reference.

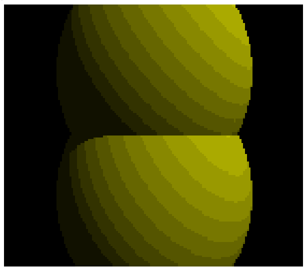
Intersection

Much as a `difference` removes the *insides* of objects, a *intersection* removes the *outsides* of objects. The result of using the `intersection` operator is that the only thing remaining is the parts which *all* the objects within the operator had in common. Let’s say that you want to make a single, colored, sugar-coated chocolate candy that won’t melt in your hands (not nameing any names). Furthermore, it must be a mathematically perfect candy. The easiest way to do this in POV-Ray is with an `intersection`, like this:

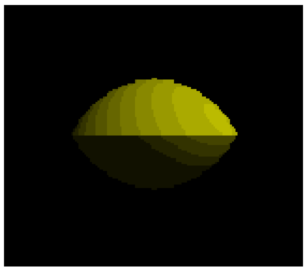
```
#include "colors.inc"

camera {
  location <0, 0, -5>
  look_at <0, 0, 0>
}

light_source { <10, 10, -10> color White }
```

```
intersection {
  sphere { <0, -1, 0>, 2 }
  sphere { <0, 1, 0>, 2 }

  pigment { color Yellow }
}
```

This code takes two spheres that overlap, like this:



Then, it uses the `intersection` operator to remove *everything* that isn't overlapping, leaving an a remarkably sweet-looking goody, like this:



Although intersections are a little more difficult to imagine than some of the other CSG operators, they can be a very powerful tool. You can find a complete reference in the intersection section of the Language Reference.

Merge

Merge is very similar to union. In fact, the only difference between the two is that, if the objects actually overlap, `merge` will make the interior a smooth, continuous unit. Now, obviously, this won't make a difference to you if your objects aren't opaque. But if you have transparent, overlapping objects in your scene, the original object boundaries will be shown if you use a `union` (or no CSG at all); to get around this, you muts use `merge`.

A complete reference for the `merge` operator can be found in the CSG Section of the Language Reference.

Inverse

Inverse is not used very often, but there are times when it must be used. *Inverse* will take your object and reverse what POV-Ray considers to be its "inside" and "outside." This will make no difference to the way your object looks, but it makes a great deal of difference to the wayyour object acts when you use it in CSG. Consider the intersection of a sphere and a box. If the sphere is inverted (by placing the keyword `invert` in its definition), then POV-Ray will take the intersection of the box and an object defined as "the entire universe *except* this sphere." If you think about it for a while (probaby a *long* while), you'll realize that that's the same as a `difference`. In other words, this:

```
intersection {
  box { <0,0,0>,<1,1,1> }
  sphere {
    <1,1,1>, 1
    inverse
  }
}
```

is the same as this:

```
difference {
  box { <0,0,0>,<1,1,1> }
  sphere {
    <1,1,1>, 1
  }
}
```

In fact, POV-Ray calculates `differences` using this same method. A complete reference to the `inverse` keyword can be found in the CSG Section of the Language Reference.

Advanced Objects

There are times when POV-Ray's geometric primitives aren't going to be enough for you. Face it, if you want to ray-trace something as complex as a human being, even CSG won't help you. In this case, there are two options left to you:

The first is to specify your object in mathematical terms. Obviously, this will only work if

1. Your object can be described by an n-dimensional polynomial in 3-space;
2. You know what the heck I'm talking about; and
3. You like pain

What we're trying to say here is that we're not about to teach you the math necessary to specify these objects, and, furthermore, we recommend against it, unless you really know what you're doing. Of course, if you'd like to read about the objects involved (namely, quadrics, cubics, quadrics and polys), the go right ahead. And if you can use them, so much the better. But if you don't have the math behind it, then don't worry about it; you'll probably sleep better at night. We have found these objects to be of limited use.

The second option you have is to use a modelling program. What a modelling program can do is generate extremely complex objects in POV-Ray by specifying them as a whole bunch of *really* simple objects, normally blobs, triangles, smooth triangles. or bicubic patches. These objects, much like the the mathematical ones above, are not generally meant for human consumption -- in other words, don't bother trying to create objects with these by hand, because unless you really know what you're doing, you’ll probably just waste a lot of time.

Instead, find a good modelling program (there are many free and shareware ones out there; try the Resource Library), create the complex object in there (usually the modelling programs will have a very nice, graphical interface) and run POV-Ray on the file it creates. You will save a lot of time and effort.

Top of Document   Main Page   Step 3: Creating Simple Scenes   Step 5: Conclusion

---

The Online POV-Ray Tutorial

POV-Ray Basics

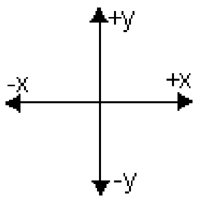
(Show Jump Points) (Hide Jump Points)

Before you can start creating scenes in POV-Ray, you need to know a few things: how to describe objects in three dimensions, some of POV-Ray's basic notation, and other stuff. This section will give you the background knowledge you'll need to get started.

Quick reference:

POV-Ray's Coordinate System

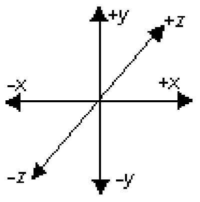
The source code file, the file POV-Ray takes as input, is really one big list of descriptions of objects. The very first thing we need in order to describe objects is a way of telling POV-Ray *where* things go. Such a method is called a coordinate system. If you have taken elementary algebra, you will already have experience with a simple coordinate system: a two-dimensional (or 2D) Cartesian coordinate system. A quick graph of a 2D Cartesian plane looks something like this:



Any position on this graph can be specified by a set of coordinates, usually written in the form `(x,y)`. The **x** coordinate corresponds to its position along the horizontal, or **x**, axis, and the **y** coordinate corresponds to its position along the vertical, or **y** axis. For example, `(0,0)` corresponds to the point in the middle of the graph, or the *origin*. The point `(1,3)` corresponds to the point on the graph one unit right from the origin, and three units up from the origin. Negative numbers can also be used: `(-6,4)` corresponds to the point 6 units left from the origin, and four units up. You get the idea.

Now this is all well and good, but when we look at things other than our computer screen, we notice we can observe *three* dimensions, not two: in other words, we describe objects not just by how far to the right (or left) and how high (or low) they are, but also how close they are in front (or in back) of you. In other words, to be able to describe a real scene to POV-Ray, we need, in addition to the **x** and **y** coordinates, a third coordinate. This coordinate is called (surprisingly enough) the **z** coordinate.

The coordinate system that POV-Ray uses, then, is called a three-dimensional (or 3D) Cartesian coordinate system. A quick graph looks like this:



(You have to use your imagination somewhat: that third axis is not a diagonal but is perpendicular to your computer screen -- imagine it shooting out at your face). As you can see, it looks similar to the 2D graph, except that one additional axis has been added: the **z** axis. Because of the additional possible direction, points in this coordinate system must be described in the form `(x,y,z)`. The point `(0,0,0)` corresponds to the origin, or center of the graph, and `(1,-2,7)` corresponds to the point one unit to the right of, two units below, and seven units behind the origin.

If you have experience with mathematical 3D coordinate systems, you will notice that the axes are labelled slightly differently than the system most commonly used in mathematical terms. The axis we have drawn above is not fixed in POV-Ray -- the way the axis looks (in terms of which axes are which) really depends on where you place your *camera* in POV-Ray. We'll get to explaining the camera soon. For now, just understand that the labels on the axes may change, depending on how you position your camera.

The 3D graph above represents a coordinate system that POV-Ray can use. Visualizing objects and scenes in three dimensions can be tricky. Often, a pad of paper and a pencil can be extremely valuable tools, especially in more complex scenes. Alternatively, you can take a look at the Resource Library for some graphical tools that may help.

Vectors in POV-Ray

POV-Ray calls the number triples that define positions *position vectors*. The term *vector* refers to any group of numbers describing a certain thing -- there are color vectors and normal vectors, for example, in addition to position vectors.

In POV-Ray, vectors are surrounded by angle brackets (that's < and >). For example, to specify the origin in terms that POV-Ray understands, we would say `<0,0,0>`.

The *magnitude* of a vector can be thought of as the "length" of the vector. Imagine a line from the origin to the point in the coordinate system represented by your vector. The magnitude is the length of this line. (If you really care about the math, the magnitute can be computed as the square root of the sum of the squares of the elements of the vector -- but don't worry, you probably won't have to know that).

An important thing to know about is a POV-Ray feature called *vector promotion*. Vector promotion is when a single number is substituted in place of a vector. The single number is then *promoted* to a vector, one with all elements equal to that number. For example, promoting the number `2` to a three-dimensional vector would result in the vector `<2,2,2>`. Vector promotion is done automatically for you by POV-Ray in most cases -- just put in a single number instead of a vector. This feature allows you to quickly specify similar vectors.

How to describe color: RGB and RGBF Vectors

Much as any position within the scene can be specified by a three-element vector, so can any color. In describing a position, each coordinate in the vector corresponds to the position along a particular axis. In describing a color, each element of the vector corresponds to the amount of a primary color -- red, green and blue. Such a vector is called a *RGB vector* (for **r**ed **g**reen **b**lue vector).

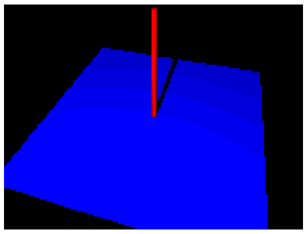
In a position vector, the individual elements can be any real number at all (actually, this isn't quite true -- there are upper and lower limits set by the hardware constraints of your computer). In a RGB vector, the numbers should be between `0.0` and `1.0`. You can have values higher that `1.0`, but they don't correspond to any physical property (what's greener than green?). A value of `1.0` means 100% of that color. For example, the color black, which is actually the absence of all color, is described by the color vector `<0,0,0>`. The color white, a complete combination of all three primary colors, is specified by the color vector `<1,1,1>`. Try experimenting with the Color Tool to find the color vectors for particular colors -- it will help you get a good "feel" for describing colors in terms of POV-Ray color vectors.

In addition to RGB vectors, you can specify a color in POV-Ray with an RGBF vector. As you might guess, a RGBF vector is like a RGB vector, but with one extra element - the F, for filter. The filter value specifies how transparent the pigment is, ranging from `0.0` (not transparent at all) to `1.0` (100% transparent). RGB vectors have an implied filter value of `0.0` -- in other words, a color specified by a RGB vector will be perfectly opaque. A filter value of `1.0` means that all light will be let through, but the light will still be filtered. For example, the RGBF vector `<1,0,0,1>` acts like red cellophane -- 100% of light is passed through, but it is filtered by the red pigment. RGBF vectors can be a little confusing at first, but they aren't too difficult once you get the hang of it.

These are the most commonly-used ways of specifying color. There are a few more ways to do it; if you want to read about them, look at the color section of the Language Reference.

Normal Vectors

Occasionally you will be called upon to specify a *normal* vector in POV-Ray. Simply put, a normal vector is a vector parallel to a given plane in three dimensions. Imagine a flat sheet of paper. If you were to poke a pencil all the way through it so that the end of the pencil was touching the paper and the pencil was standing straight up (with respect to the paper), the pencil would represent the normal vector to the paper. In the picture below, the normal vector is in red and the plane is in blue.



Note that the magnitude of normal vectors is not important (as long as it is non-zero). This is because normal vectors are used to specify an *orientation*, not a distance.

POV-Ray is kind enough to automatically define three normal vectors for you: **x** (corresponding to `<1,0,0>`), the normal vector for a plane lying along the y and z axes, **y** (corresponding to <0, 1, 0>), the normal vector for a plane lying along the x and z axes, and **z** (corresponding to <0, 0, 1>), the normal vector for a plane lying along the x and y axes. Any time you are asked for a normal vector (or any vector, really) you can substitute those letters.

POV-Ray Source Code

Source code is the name for the text you give to POV-Ray. POV-Ray reads the source code file and outputs an image. There are two things you need to know about POV-Ray source code:

1. POV-Ray source code is case sensitive
2. POV-Ray ignores whitespace
3. Ordering is unimportant

Case sensitive means that upper and lower-case letters are not treated as the same by POV-Ray. For example, to POV-Ray, `sphere` is not the same as `Sphere` and is not the same as `SpHeRe`. *Whitespace* is the common name for any characters you can't directly see on screen -- spaces, tab characters (the invisible characters put there when you press the **Tab** key), carriage returns and line feeds (the invisible characters put there when you hit the **Enter** key). Between any two words or symbols in your source code, POV-Ray doesn't care whether you put one space, two spaces, one hundred spaces, a new line, or any other whitespace.

For example, the phrase:

```
one two
```

the phrase

```
one                two
```

and the phrase

```
one
two
```

are all treated the same by POV-Ray.

Ordering means the order in which you declare objects. POV-Ray does not care where in the file the objects are -- it makes no difference to the final scene. (VRML programmers will note that this is a very different approach than VRML's "virtual pen" concept). This does not hold entirely true for some attributes and CSG operations (both of which we will describe in detail later), but in the outer-most level in POV-Ray (the one in which you list the objects in your scene) it doesn't matter.

Comments in POV-Ray Source Code

Comments are another useful part of POV-Ray source code. A *comment* is a portion of text that POV-Ray will ignore. It is used to add information to the source code, usually to make things clearer to the human reader. Comments can be enclosed in `/*` and `*/`, or, for single-line comments, can be prefixed with a `//`. For example:

```
// this is a single-line comment
/* this is
   another comment.  it can be as long as you want it to be */
```

C and C++ programmers will recognized this comment style. For a detailed description of comments, see the comments section of the Language Reference.

Including files

Including files is a feature of many languages that makes re-using code easier. If you have, for example, many red objects in your scene, you will find it cumbersome (and not very readable) to type the correct RGB vector for red every time. POV-Ray comes to the rescue with a file full of pre-defined colors, which you can use and re-use in your source code. (POV-Ray also comes with files of textures and even objects; we'll get to those later). You can take advantage of these files by adding the string `#include "`*`filename`*`"` to the beginning of your file. For example, to use the pre-defined colors, you would add the string

#include "colors.inc"

to the beginning of your file. Technically, the statement does not have to occur at the beginning of the file, but the convention is such, and it makes for readability.

The example statement above tells POV-Ray to look for the file called `colors.inc` and to read it before continuing to the rest of your file. `colors.inc` defines many colors, such as `Red`, that you can use in your file any time you need a color, in place of a RGB (or RGBF) vector. This makes your source file *much* easier to read. Later in the tutorial, you will learn how to define your own colors (and objects, and textures, and so on) and how to put them in your own text files. For now, know how to use the provided ones and be happy.

Now that you've got *that* out of the way, you're ready to start creating your first scene... almost.

Top of Document Main Page Step 1: Introduction to POV-Ray and Ray-tracing Step 3: Creating Simple Scenes

---

The Online POV-Ray Tutorial

Conclusion

(Show Jump Points) (Hide Jump Points)

Well, well, well. You've come far from the beginning of the Path of Learning. We'd just like to take a monute to congratulate you on your new found skills. Congratulations. You now know the POV-Ray language. Anything that can be done in POV-Ray is now within your grasp. However, this is just the beginning of your full journey through POV-Ray. This tutorial has helped as much as it can. Now all you need is experience. The main thing that distinguishes a good POV-Ray artist from an awesome POV-Ray artist is experience with ray tracing and possibly better hardware, but that's not as important. There are a great many ray tracing libraries on the internet and many of them include source. Check 'em out and see how the masters do things. Then you will be well on your way to becoming a POV-Ray master yourself.

Top of Document Main Page Step 4: Advanced POV-Ray Features

The Online POV-Ray Tutorial

Introduction to POV-Ray and Ray-tracing

(Show Jump Points) (Hide Jump Points)

In this section, you will learn what ray-tracing is, how it works, what POV-Ray is, and how POV-Ray relates to the rest of the ray-tracing world. This is a good section if you’ve never experimented with (or even heard of) ray-tracing before.

Quick index:

What is ray-tracing?

Ray-tracing is a method of creating visual art in which a description of an object or scene is mathematically converted into a picture. In more precise terms, ray-tracing is the process of mathematically generating near-photorealistic images from a given description of a scene via geometrical modeling of light rays.

Ray-tracing can generate very beautiful and complex scenes, and can open exciting possibilities as a new method of creating visual art. One of the most important advantages of computer-based ray-tracing over more "orthodox" art forms is that it removes the need for technical skills (such as the ability to paint, draw, or sculpt) that may take years to master, and places the burden on the computer. This leaves the user to be as creative as possible, without having to spend years learning difficult skills. In fact, you should be well on your way to creating some exciting pictures before you’re far along this tutorial.

Ray-tracing can require millions and even billions of complex mathematical calculations and, as such, is usually done by computer (and even then, is not a speedy process). Usually, in the computer-based ray-tracing procedure, a file containing the description of a scene (in terms that the ray-tracing software can understand, and usually in some human-readable format) is converted, by the computer, into an actual image of the scene. Later on in the tutorial you will learn how to use the POV-Ray software to create scenes of your own, with the help of your computer.

Of course, ray-tracing is not a magical technique that makes all art easy. Certain types of scenes are difficult or impossible to create with ray-tracing software. In most cases, you will find that ray-tracing is good at generating mathematically simple objects, such as those composed of spheres, cones and cubes, and poor (slower or more difficult to describe) at generating more complex objects, like a human face. There are several ways of getting around these barriers, but ray-tracing, like all forms of art, does have certain inherent limitations.

Because ray-tracing is based on math, ray-traced scenes have some distinct characteristics. For example, in a simple ray-traced scene, all objects are in focus, and all shadows are crisp and well-defined; in fact, every object in the scene is mathematically perfect. (When was the last time you saw a mathematically perfect pear?) Because of this, images produced by ray-tracing tend to look slightly odd, or even surrealistic, a side-effect many artists can exploit to their benefit. It is, interestingly enough, considered a mark of a "true ray-tracing artist" to be able to create more realistic, "less perfect" scenes.

How does ray-tracing work?

We won’t go into all the gory details, but having a general understanding of what’s going on behind the scenes (so to speak) can be helpful when you start ray-tracing your own images. Although there are several methods of ray-tracing, one of the most common (and the one the POV-Ray software package uses) works something like this:

First, an internal model of the scene is generated, with your computer screen included as the receiving "eye" in the model. Then, the software traces imaginary light rays backwards from where their endpoint lies (a pixel on your computer screen) to their initial point (some light source in the scene). This step is repeated, pixel by pixel, until the entire image has been created.

The reason the software traces the light rays backwards, instead of starting at the light source, is for efficiency’s sake -- if a light ray doesn’t end up on your screen, then you, as the user, don’t care about it (because you’ll never see it). By tracing the light rays backwards, beginning at the computer screen, the software can assure that every light ray it calculates *is* one you care about, because it knows that it will end up on your screen.

In their journey, the light rays can be reflected by mirrors, refracted by glass, or undergo various other contortions, all of which result in a single pixel of the final image. Because the ray-tracing software must trace one ray of light for each pixel in the output image, and because the light rays can undergo so many contortions, the process of ray-tracing can take a very long time, depending on the size and complexity of the image and the processing power of your computer. Unless you have an extraordinarily fast computer, you will most likely find yourself waiting around impatiently for your scene to finish rendering once you’ve begun to ray-trace in earnest.

What is POV-Ray?

POV-Ray is a high-quality, freely available ray-tracing software package that is available for PC, Macintosh and UNIX platforms. Yes, that’s right, it’s free! If you’re a programmer interested in POV-Ray, you can also pick up a copy of the source code without charge. POV-Ray is perhaps one of the most commonly used ray-tracing software to date, because of its relative ease of use, cheap price, and high quality.

POV-Ray is no toy. Despite not generating any direct income from their POV-Ray software, the POV-Ray Team (the people responsible for POV-Ray) has managed to create a commercial-quality product and, in the true spirit of the Internet, distribute it widely and without charge. As a consequence, POV-Ray is one of the most popular ray-tracing programs to date. We, the authors of the *Tutorial*, think that POV-Ray is one of the greatest things to come out of the Internet (gee, is it a little obvious?), which is why we decided to create this Web page.

At any rate, POV-Ray is what is known as a "rendering engine". What this means is that POV-Ray will take a file as input and generate an output file, but does not have much in the way of interface. There are modellers available for POV-Ray that will do that kind of visualization for you, if you choose, but we recommend you only start using those tools once you have a firm grasp of the POV-Ray language. Otherwise, you'll get stuck down the road.

Describing scenes to POV-Ray is fairly simple. You give POV-Ray a file containing a description of every object in the scene, written in the POV-Ray language (which you will learn later in the tutorial). Each object's description consists of:

1. What type of object you want (one of POV-Ray's simple objects or one you've created yourself); and
2. Various attributes of the object (its color, how it reflects light, etc).

POV-Ray takes this file and generates a picture, which you can then view.

The *Path of Learning* is not meant to cover every single detail of the POV-Ray language. Such a task would be tremendous (although the Language Reference has come very close). Instead the *Path* will guide you as you learn about each section, and, when appropriate, refer you to more complete sources of information. At the end of the *Path*, you should be well on your way to being a certified POV-Ray blackbelt.

How do I set up POV-Ray?

The latest version of POV-Ray can always be found at ftp.povray.org. You will have to download the correct version for your computer (there are versions available for most operating systems) and to set it up.

Once you have POV-Ray, how you set it up is highly dependant on your operating system. We're not about to teach you how to use your own computer; if you can't set it up yourself, ask a local computer guru to help.

As we mentioned above, POV-Ray doesn't have much of an interface; on most operating systems, you will give POV-Ray the name of your input file, the name of your output file, and a whole bunch of other options via the command line. You will also need some form of image viewer and/or converter in order to display the output files that POV-Ray creates; again, this is highly operating-system-dependant.

POV-Ray also comes with documentation and example scenes; these make excellent references if you're stuck or need to know more.

Ok, we're ready to start learning the *real* stuff now!

Top of Document Main Page Step 2: POV-Ray Basics

Welcome to

If this is your first visit, you may want to start on the Path of Learning below. Or, if you are interested in specific information, you may want to browse through the Exploration Toolkit, Reference Shelf and Libraries below. You can also read about the *Tutorial*'s design.

If you're not sure where to find what you want, try the Help Desk.

NEW!: A word about the release of POV-Ray 3.0.

The Path of Learning

The *Path of Learning* is meant to lead you, step by step, from a ray-tracing novice to a full-fledged POV-Ray expert. If you are a beginner, we recommend you start at Step 1. If used other ray-tracing programs before, or just can't wait any longer, jump ahead to step Step 3.

Step 1: Introduction to POV-Ray and Ray-tracing
: **What is ray-tracing? What is POV-Ray?** If you've never done any ray-tracing before, this is a good place to start reading. It provides a brief overview of what ray-tracing is, how it works, and how it's used. This section will also tell you what exactly POV-Ray is, how it relates to the rest of the ray-tracing world, where to find it and how to set it up.

Step 2: POV-Ray Basics
: **What do I need to know before I start?** This section covers all the vital information you need to know before you can start creating your own scenes in POV-Ray, including the mathematical background necessary (yes, there is a little) and information about source code.

Step 3: Creating Simple Scenes
: **How do I get started?** This section covers the basics of scene creation in POV-Ray. If you're

anxious to get going, start here.

Step 4: Advanced POV-Ray Features

> **What else can POV-Ray do?** This section covers some of the more powerful (and complex) elements of the language, including advanced objects, attributes, lighting techniques and camera options.

Step 5: Conclusion

> **What now?** A few words of advice, and some congratulations on your new-found skills. The (sigh) end of the *Path of Learning*.

Reference Shelf

If you need quick access to specific information, one of the resources below may be able to help you.

POV-Ray Language Reference

> A searchable glossary of all keywords and directives of the POV-Ray language. Covers the meaning, syntax and usage of all POV-Ray terms.

Glossary / Index

> A combination glossary and index containing ray-tracing, POV-Ray, and other technical terms, as well as references to the *Tutorial* and *Language Reference*.

Command-line Parameter Reference

> A complete description of all command-line parameters that POV-Ray takes, what they do, and how to use them.

Tips and Tricks

> Tips and tricks covering the practical and the artistic sides of ray-tracing. Find out how to save yourself time in ray-tracing, as well as what makes your scene look realistic, artistic and, of course, cool.

Libraries

The libraries below contain useful bits of information. You can browse through each of them, or even submit your own for the rest of the world to use.

Texture Library

> A library of included and contributed textures. Contains source code and sample images for all

textures included with POV-Ray releases, as well as textures contributed by other readers.

Object Library
: A library of built-in, included and contributed objects. Contains source code and sample images for all non-primitive objects included in POV-Ray releases, as well as objects contributed by other readers.

Scenes Library
: A library of nifty scenes, with source code! Feel free to contribute your own, if you think you’ve got something worth sharing.

Resource Library
: A library of contributed POV-Ray resources: web pages, programs, and anything that can be of use to POV-Ray artists.

Exploration Tools

These tools are provided in order to give you a quick and accurate "look-and-feel" of many of POV-Ray’s features. Note that if your web browser does not support forms, Javascript or Java, some of the tools may not work.

Color Tool
: Displays the color corresponding to a particular RGB vector. The vector elements are selected by the user. Helpful in finding the right color for an object.

Normal Tool
: An explorable interactive graphical demonstration of all the normal attributes: bumps, dents, ripples, waves, and wrinkles.

Finish Tool
: An explorabale interactive graphical demonstration of all the finish attributes: ambient, brilliant, crand, diffuse, phong, reflection, refraction, roughness, specular.

Top of Document

The Online POV-Ray Tutorial

Creating Simple Scenes

(Show Jump Points) (Hide Jump Points)

The POV-Ray language is fairly easy to use, once you understand it. In fact, if you have any experience with programming, you will find POV-Ray very easy -- there are no variables, conditionals, loops, or anything else that can make programming tricky. Basically, a POV-Ray source file (the file you make and give to POV-Ray) is just a list of objects and their descriptions. Of course, describing the scene you have in your mind to POV-Ray is the tricky part, because you have to speak POV-Ray's language.

Quick index:

Creating simple objects

The building blocks of all POV-Ray objects and scenes are called *primitive*s. Primitives are objects that POV-Ray already knows about, and all you have to do is describe a few attributes. POV-Ray primitives are usually simple geometric shapes such as spheres, cubes, and cones.

Describing primitives, in general, take this form in POV-Ray:

```
Object_Name {
  Object_Parameters
  Some_Simple_Attribute
  Some_Other_Simple_Attribute
  Some_Complex_Attribute {
    Some_Attribute
    Some_Other_Attribute
  }
}
```

This isn't very enlightening. Let's take a look at a short example:

```
sphere {
  <0, 0, 0>, 5
  pigment {
    color rgb <1, 0, 0>
  }
}
```

Deciphering what this does isn't too tricky. The code defines a sphere with its center at the origin (that's `<0,0,0>`, remember?) and with a radius of 5 (in other words, the distance from the center of the sphere to any point on the edge of the sphere is exactly 5 units). The phrase `pigment { color rgb &lt1,0,0> }` simply means that the sphere's pigment (or color) attribute is described by the rgb vector `<1,0,0>`, which is the color red. You could have just as well used `color Red`, if you had `#included` the correct file. The pigment attribute, by the way, is a complex attribute, of which *color* is just one of the many attributes that can go inside it.

There are two types of primitives in POV-Ray: *finite* primitives and *infinite* primitives. *Finite* primitives have well-defined limits. Examples of finite primitives include spheres, cones, torii, and blobs. *Infinite* primitives have components that can potentially stretch to infinity -- for example, a plane is both infinitely thin and infinitely wide. Examples of infinite objects include planes, quadrics and cubics. At any rate, describing primitives in POV-Ray is only a matter of knowing the syntax for the particular primitive you want to describe. You can find a complete syntax reference in the finite object and infinite object language references.

By now, you're probably itching to make your first scene. Before you can do that, however, you need to learn about two things: the camera and light sources.

The Camera

Before POV-Ray can generate the scene, it needs to know from where you are looking. If you imagine your computer screen as the camera taking a snapshot of the scene you're describing, you'll see POV-Ray needs to know a) where the camera is in the scene, and b) which direction it's pointing. Such data is given to POV-Ray through the *camera* object. As you might imagine, the camera object is a rather important one: in fact, POV-Ray requires that there be one and only one in each scene.

There are many attributes that the camera object can have; of these, we will only concentrate on the two most useful: the *location* and the *look_at* attributes. A complete reference of all the camera attributes can be found in the Camera Reference.

A simple camera in POV-Ray looks like this:

```
camera {
  location <2,5,-10>
  look_at <0,0,0>
}
```

This example defines a camera located at `<2,5,-10>` and pointing at the origin. This means that anything with a **z** coordinate less than `-10` will *definately* be invisible -- it will be behind the camera!

You can put the camera anywhere you want in the scene, including inside of objects (although you may not see very much), with one exception: you may not place the camera directly over the origin and have it looking straight down. For complex mathematical reasons, this will cause POV-Ray to generate an error. If you need that type of setup, position the camera a little to the left or the right -- your problem will be solved, and your scene will look (almost) exactly the same.

Anyways, now that we have a way of receiving light, we need to have a way of providing light.

Let there be light! (Light sources)

If you gave POV-Ray a file containing the camera definition above and the sphere definition before that, the output image would be a lovely blank picture. This would happen because you'd have no light in your scene. To add light (thereby enabling you to actually *see* something), you need to add a *light source*.

There are a few different types of light sources in POV-Ray. We will concentrate here on the most simple (and useful): the *point light source*. A point light source can be thought of as an infinitely small object that emits light. Because they are infinitely small, point light sources cannot be directly seen (so you don't have to worry about them appearing in your scene). However, their *effects* can certainly be seen: your scene lights up!

Point light sources as known as non-attenuating light sources: the emitted light does not get weaker with distance. This means that you can illuminate your entire scene with one point light source placed far away from the scene. You can have as many light sources as you want, but they are computationally expensive -- the more you have, the longer POV-Ray will take to trace your scene.

An example of a simple point light source definition in POV-Ray looks like this:

```
light_source {
  <0,10,-10>
  color rgb <1,1,1>
}
```

The first vector is a position vector specifying the location of the light source. The second vector specifies the color (and brightness) of the light. It is generally a good idea to use white or gray light, as using colored light can have side effects that are not immediately obvious (for example, green objects will not show up when exposed to pure red light). Complete information for light sources can be found in the lights section of the Language Reference.

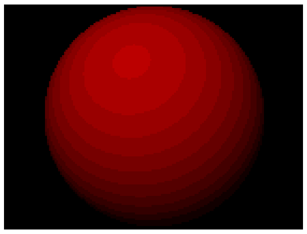
Anyways, now that we can add light, we're ready to construct our first full scene.

The first example scene

Putting together all we have learned to far, we get a complete POV-Ray source code file that looks like this:

```
// This is a simple red sphere

// first, the camera position
camera {
  location <2,5,-10>
  look_at <0,0,0>
}

// now, some light
light_source {
  <0,-10,0>
  color rgb <1,1,1>
}

// the sphere
sphere {
  <0,0,0>, 5
  pigment { color rgb <1,0,0> }
}
```

After running POV-Ray, the output image looks like this:

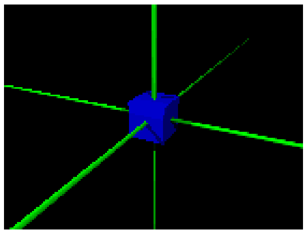


Finally! Your first image! Of course, this one is a little boring -- but don't worry, we'll get to some fun stuff soon. For now, experiment! It's the best way to learn. Try replacing the sphere with other objects and seeing what happens. The objects that you should easily be able to use are boxes, cones, cylinders, spheres, torii and planes.

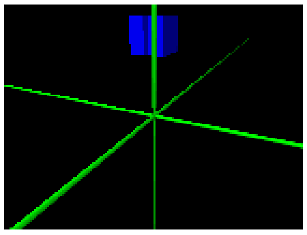
Transformations

So now we can create some simple objecs. But wait! Some of these objects can only be created around the origin (like the torus). What if we want to put them somewhere else? What if we want to move them around? POV-Ray provides answers to all these questions in the form of *transformations*. Transformations, in ray-tracing terms, are attributes that change the position, size or orientation of objects (and of the various attributes of the objects). The most common types of transformations, and the ones that POV-Ray supports, are *translations*, *rotations* and *scalings*.

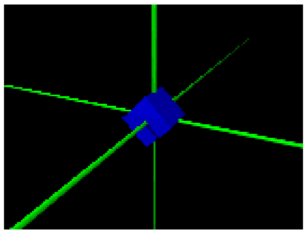
A *translation* is a transformation that moves an object relative to its current position. It is specified in POV-Ray by the phrase `translate <`*`x`*`,`*`y`*`,`*`z`*`>`. Translations are easy to visualize. Consider a cube sitting on the origin, like this:



Our camera is positioned so that the x axis increases to the right, the y axis increases upwards and the z axis increases towards us. A translation of `<-1,4,2>` results in the cube being moved left one unit, up four, and back two, like this:

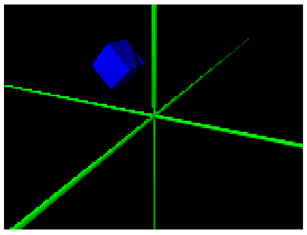


A *rotation* is a transformation that changes the orientation of an object (the way that it's facing). Rotations are the most complex of the transformations. They are specified to POV-Ray by the string rotation <*x*,*y*,*z*>, where *x*, *y*, and *z* are the number of degrees (not radians) around the respective axis. Consider the original cube up above. A rotation of `<0,0,45>` rotates the cube 45 degrees around the **z** axis, leaving us with a cube looking like this:

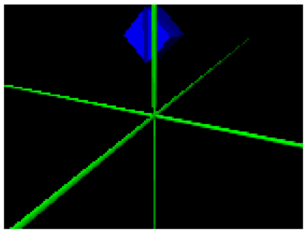


A quick way to remember which way the objects are going to rotate is by usings the so-called "left hand rule." Hold out your left hand, fingers clenched and thumb out. Point your thumb in the positive direction of the axis you are rotating about (if you're rotating about more than one axis at a time, this won't help you -- unless you have more than one thumb!) The direction that your fingers curl is the direction an object will rotate when the number of degrees is positive. (Negative degrees rotate the opposite direction).

Another important thing to remember about rotations is that they are always with respect to the coordinate axes -- in other words, unless your object is located at the origin, it will orbit around the axis (or axes) you are rotating it about. For example, this is what would happen if we translated the cube first, and then rotated it:

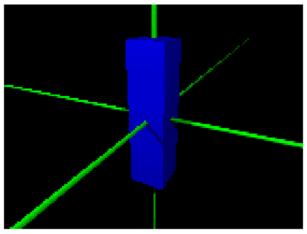


To get around this, make sure you `rotate` your object when its centered at the origin, and *then* `translate` it. Your picture will end up like this:



Transformations are one of the few aspects of POV-Ray in which the order matters, simply because transformations are always made with respect to the object’s current orientation.

The last translation you need to know about is *scaling*. Simply enough, scaling changes the size of the object with respect to its current size. Scaling is specified in POV-Ray via the string `scale <x,y,z>`. The elements of the vector specify the how much to scale the shape with respect to the coordinate axis: a scale of `1.0` leaves the object the same, and a scale of `0.0` or less is invalid. Going back to our original cube, if we scaled the object with the string `scale <1,4,1>`, we would get a result like this:

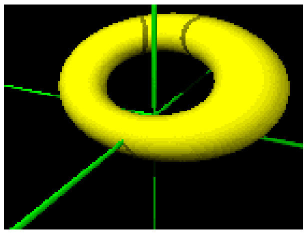


Because of *vector promotion* (if you don’t remember what that is, you can re-read about it), scaling can also take a single number rather than a vector. This causes the object to be scaled in every direction by that number. For example, the phrase `scale 2` is the same as the phrase `scale <2,2,2>`.

Transformations are placed like any other attribute. For example:

```
torus {
  3, 11
  pigment { color Yellow }
  scale <1.5,1,1>
  rotate <-45,0,0>
  translate <0,2,0>
}
```

This code makes a yellow torus, slightly widened around the x axis, rotated -45 degrees around the x axis and with its center at `<0,2,0>`, like this:



Note that torus objects are created around the origin, so you are in fact *forced* to use transformations to get them where you want... luckily for you, you now know how. And to quote G. I. Joe, knowing is half the battle.

Texture

We admit it -- we lied to you. The `pigment` attribute is actually a part of a bigger attribute called the `texture` attribute. Every time you used `pigment`, it should have really looked like this:

```
texture {
  pigment { color Red }
}
```

The reason that POV-Ray is a little loose about the `pigment` attribute and lets you use it outside of `texture` is because `pigment` is so frequently used by itself that it becomes a pain to type out the whole `texture` statement. In fact, most parts of the `texture { }` block you can do the same thing with. Either way, they have the same effect.

The `texture` attribute contains attributes describing the outward appearance of the object: `pigment`, `finish` and `normal`. The `pigment` attribute, as you know, describes the color of the object (although it's a *lot* more complicated than what we've shown you so far). The `finish` attribute describes how the object "interacts with light" -- highlighting, metallic luster, shinyness, reflectivity, etc. The `normal` attribute describes some three-dimensional features of objects, such as bumps, waves, and ripples. We'll cover these one by one.

Pigment

You've seen the use of the `color` attribute within the `pigment` attribute (for example, `pigment { color Blue }`). A more complete description that what we've given you so far can be found in the Color section of the Language Reference. A more flexibe attribute, however, is `color_map`. `color_maps` are used to do a wide variety of things. Basically, a `color_map` defines bands of color on a "map" ranging from `0.0` to 1.0 Let's look at a simple example:

```
color_map {
  [0.0  color Red]
  [0.25 color Blue]
  [0.9  color Green]
}
```

This defines three bands of color: red from `0.0` to `0.25`, blue from `0.25` to `0.9`, and green from `0.9` to `1.0`. The other commonly used format looks like this:

```
color_map {
  [0.0 0.25 color Red]
  [0.25 0.9 color Blue]
  [0.9 1.0  color Green]
}
```

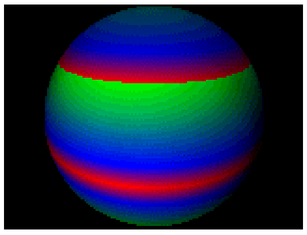
They both do the same thing; the second one just contains information about where you want the bands to stop as well as start.

The next step is tell POV-Ray what to *do* with this. This is done by using of the many *pigment types*. A simple pigment type is called `gradient`. Gradient creates bands of color based on the color map. Using the source code from the first scene we created, and replacing the color Red with our color map and pigment type, we get this:

```
sphere {
  <0,0,0>, 5
  pigment {
    gradient <0, 1, 0>
    color_map {
      [0.0 color Red]
      [0.25 color Blue]
      [1.0 color Green]
    }
  scale 3
  }
}
```

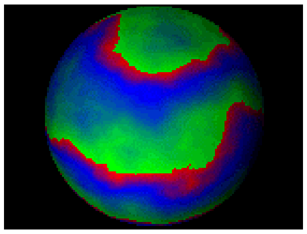
This source code requires a bit of explaining. The vector following the `gradient` keyword is the normal vector to the orientation of the bands of color (you remember normal vectors, don't you? Or did you think we were wasting our time telling you stuff you didn't need to know? Admit it! You skipped over that section! Well, we're forgiving; you can go back and read about it *again*). The `scale` statement applies to the *pigment*, not to the object (look carefully at where it's placed -- inside the `pigment { }` block).

Our sphere now looks like this:



A careful examination of this image yields some interesting facts. Starting from the top down, you can see a slight bit of green (the rest of it was cut off), which fades into the the large blue band, which in turn fades into the small red band. The red band is abruptly cut off and the cycle repeats itself again. However, the next time, the pattern has reversed! The red band is on the top. This is because `gradient` patterns reverse themselves at the origin. To get around this, you can translate the texture away from the origin (you can apply all transformations to textures, remember?). More information on `gradients` can be found in the gradient section of the Language Reference.

Ok, now let's try something else. Add the phrase `turbulence 0.5` after the `gradient` statement. The resulting picture looks like this:

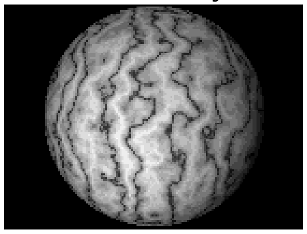


Whoah! The `turbulence` keyword, as you may have guessed, "mixes stuff up." With this color map, we get a freakish plasma-like sphere. Values for `turbulence` range from `0.0` to `1.0`. A complete description can be found in the turbulence section of the Language Reference.

There are many other pigment types than `gradient`. For example, there is a pigment type called `marble`. By itself, rather boring and un-marble-like. However, with a high turbulence, it can create some very realistic marble pigments. Here's some sample source code:

```
sphere {
  <0,0,0>,5
  pigment {
    marble
    turbulence 1  // full turbulence
    color_map {
      [0.0 color Gray90] // 90% gray
      [0.8 color Gray60] // 60% gray
      [1.0 color Gray20] // 20% gray
    }
  }
}
```

This high-turbulence marble pigment generates some very *nice*-looking marble:



Not too shabby, huh? Other pigment types include wood, agate, bozo, and a host of others that can be found in the pigment section of the Language Reference. And although technically not pigment types per se, you may want to check out the checker and hexagon pigment patterns, as well as the image map pattern (which lets you map an external image to an object), all found in the same section as above. And remember, the best way to learn is to experiment!

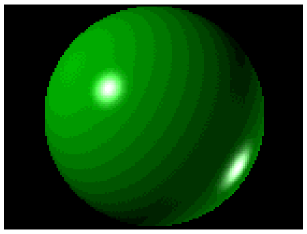
Finish

Finish describes how the objects interact with light: how much they reflect, how they shine, how metallic they are, etc. All finish attributes are enclosed in a `finish { }` block.

Perhaps the most used of the finish attributes is the `phong` attribute. A phong is a highlight, or glare. It is specified, strangely enough, by the `phong` attribute, followed by a number between `0.0` and `1.0` that specifies how bright the phong is. There is also a `phong_size` that controlls how "tight" the phong is -- in other words, the higher this number, the smaller in size the phong is (this is a little misleading, yes). Here we have a green sphere with a phong highlight of `1.0`:

```
sphere {
  <0,0,0>, 5
  pigment { color rgb <1,1,0> }
  finish { phong 0.8 }
}
```

When lit by two light sources, the sphere looks like this:

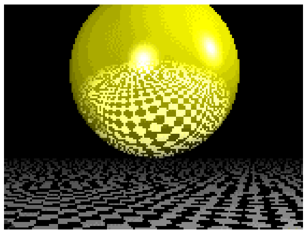


As you can see, the phong adds a nice bit of realistic "shine" whenever a light source directly hits part of the object. A more complete description of `phong` can be found in the phong section of the Language Reference.

Another finish attribute that can produce stunning effects is the `reflection` keyword. This causes objects to reflect their surroundings to a certain degree. Reflection takes one number, ranging from `0.0` to 1.0, that specifies how reflective the object is. Let’s take a look at a more complex scene with a reflective object.

```
#include "colors.inc"

camera {
  location <-2, 3, -10>
  look_at <0, 5, 0>
}

plane { // the floor
  y, 0  // along the x-z plane (y is the normal vector)
  pigment { checker color Black color White } // checkered pattern
}

sphere {
  <0, 5, 0>, 2
  pigment { color White }
  finish {
    reflection 0.9
    phong 1
  }
}

light_source { <10, 10, -10> color White }
light_source { <-10, 5, -15> color White }
```

The image this produces is:



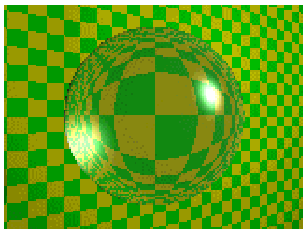
As you can see, this generates a yellowish mirrored sphere floating above an infinite checkerboard -- a variant of one of the standard ray-tracing scenes. A more in-depth description of reflectivity can be found in the reflection section of the Reference manual.

The final attribute of the `finish` keyword we will describe here is the `refraction` keyword. *Refraction* is what happens when light rays passing through a translucent object get bent, causing a distortion of everything seen through the object. For example, if you look through a crystal ball, you will see a distorted view of whatever is behind it.

The `refraction` keyword takes one value. This value should either be `0.0` or `1.0`, for refraction off and on, respectively. Although you can specify values in between, it is not recommended as it does not correspond to any known physical property. How noticeably it refracts is controlled by the `ior` keyword (for **i**ndex **o**f **r**efraction), which takes a number greater than 0. The default `ior` of "empty space" is defined as `1.0`. So, if we wanted to create the crystal ball described above, we would use something like this:

```
sphere {
  <0,5,0>,2
  pigment { color rgbf <1,1,1,.8> }
  finish {
    reflection 0.1
    refraction 1.0
    ior 1.5
    phong 1.0
  }
}
```

Remember your RGBF vectors? A filter value of `1.0` would mean this was an invisible sphere, certainly not what we want. Our filter value of `0.8` gives the sphere enough definition to be visible. The image generated looks like this:



Now we start seeing some of the true power of ray-tracing. The warped look of the checkboard pattern is due to the refraction, the bright hightlighting is due to a phong, and a bit of reflection makes this all the more realistic. Tinting the glass would be easy: just change the color of the sphere from `<1,1,1>` (or white) to whatever color you want it tinted. Modify the filter value to make the ball more and less translucent. It’s fun!

There are many other `finish` attributes that you can play with, including metallic, ambient, and crand. We’ve touched on a few; for a complete reference, read the finish section of the Language Reference. To get a good feel for most of the finish attributes, you can experiment with the Finish Tool.

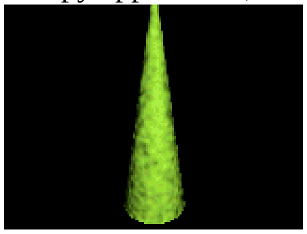
Normal

The `normal` attribute creates some simple 3D features on your objects: bumps, ripples, waves, and the like. It does not actually change the object; instead, it changes slightly the way light bounces off the object and essentailly fools the eye into believing the object is a little different than it really is. As such, the effects are not 100% true to real life, but they are much, much faster than actually describing the changes individually would be.

Let’s try a bumps example. Bumps are created with (oddly enough) the `bumps` keyword, followed by a single number, generally between `0.0` and `1.0`, that specifies the relative size of the bumps. Here’s some source code:

```
cone {
  <0,-3,0>,1
  <0,3,0>,0.1
  texture {
    normal {
      bumps 1/2
      scale 1/6
    }
    pigment { color rgb <.5,.7,.2> }
  }
}
```

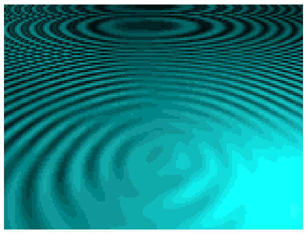
This creates a green cone with a slightly bumpy appearance, like this:



Not to difficult, eh? Imagine how difficult it would be to model all those bumps yourself. Now, here’s a fun one to try -- ripples:

```
plane {
  y, -2
  texture {
    pigment { color rgb <.1,.9,.9> }
    normal {
      ripples 0.5
    }
  }
}
```

The number following the `ripples` keyword specifies, again, the relative size of the ripples. The image this produces is:



Pretty nifty! The `ripples` keyword and its close relative, the `waves` keyword, can take a few modifiers that give a little more control than we've shown you. A complete reference can be found in the ripples section of the Language Reference. More `normal` attributes, such a dents and wrinkles, can be found in the normal section of the same document. You can also experiment with the Normal Tool to get a feeling for the various attributes.

Including Textures

Much like you learned how to include colors beforehand, you can also include textures. POV-Ray comes with a file full of some very good textures, called `textures.inc`. Including this is the same as before:

```
#include "colors.inc"
#include "textures.inc"
```

Note that you must include `colors.inc` before you include `textures.inc`, because `textures.inc` uses colors from `colors.inc`.

Using an included texture is easy. To make a sphere that uses the Jade textures, for example, you would say:

```
sphere {
  <-2, 4, 6>, 5.6
  texture { Jade }
}
```

Look through the file `textures.inc` for a list of the textures included. You can also look through `colors.inc` for a list of the colors in there.

Well, if you've managed this far, you're in good shape. Keep it up! The next section gets in to the really fun stuff.

Top of Document   Main Page   Step 2: POV-Ray Basics   Step 4: Advanced POV-Ray Features

JPEG AND MPEG STANDARDS

Abdou Youssef

Back to Top

1. Motivation for Standards

- Why Standards?
  - Compatibility

    - Production cost reduction
    - Triggering growth and product development

- Do JPEG/MPEG standards kill research?
    - Not necessarily
    - Those standards are very flexible regarding the encoder design, leaving much room for improvement
    - The trends are toward even greater flexibility in future generations of the standards
    - Another growth area is the development of systems which integrate components that use the standards

Back to Top

2. Image/Video Compression Standards (Outline)

- JPEG
    - Baseline JPEG
    - Extended JPEG
    - Lossless JPEG (DPCM + Huffman/Arithmetic)

- MPEG
    - History (H.261 and H.263)
    - MPEG1
    - MPEG2
    - Why not MPEG3
    - MPEG4

Back to Top

3. The JPEG ‘‘Toolkit’’

- JPEG provides a ‘‘toolkit’’ of techniques for compressing continuous-tone, still, color and monochrome images

- Baseline JPEG provides a DCT-based algorithm, and uses run-length encoding and Huffman coding

- Baseline JPEG operates only in sequential mode, and is restricted to 8 bits/pixel input images

- Extended JPEG offers several optional enhancements:
    - 12-bit/pixel input
    - Progressive transmission
    - Choice between Arithmetic and Huffman coding

        - Adaptive quantization
        - Tiling
        - Still picture interchange file format (SPIFF)
        - Selective refinement

    - Applications of JPEG
        - Desktop publishing, color fax, photojournalism, medical images, general image archiving systems, consumer imaging, graphic arts, and others

Back to Top

4. The Baseline JPEG Algorithm

1. It operates on 8×8 blocks of the input image

2. Mean-normalization (subtract 128 from each pixel)

3. Transform: DCT-transform each block

4. Quantization
    - An 8×8 quantization matrix Q is user-provided
    - Each block is divided by Q (point by point)
    - The terms are then rounded to their nearest integers

    - Remark: Up to 4 quantization matrices per image are allowed (for example, one for luminance, and for each of the three color components)

5. Entropy-coding of the DC coefficients (the top left coefficient of each quantized block) using DPCM+Huffman
    - Huffman-encode the DC residuals derived from the difference between each DC and the DC of the preceding block

6. Entropy-coding of the AC (i.e., non-DC) coefficients
    - Zigzag-order the quantized coefficients of each block
    - Record for each nonzero coefficient both its distance (called run) to the preceding nonzero coefficient in the zigzag sequence, and its value (called **level**)
    - Huffman code the [run,level] terms using one single Huffman table for all the AC's of the image

Back to Top

5. The Quantization Matrices

- They are user-provided

- They can be computed using the contrast sensitivity function of the HVS
- Their values are 8-bit integers
- They provide control over the bitrate by scaling them by a constant factor
- Example of Q:

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Back to Top

6. The DC Huffman Table

- The DC residuals are in the range [-2047,2047]
- Divide this range into 12 categories corresponding to ‘‘one-dimensional ripples’’
- Category k, k=0,1,...,11, consists of 2^k integers ranging from -2^k+1 to -2^{k-1}, and from 2^{k-1} to 2^k-1
- Develop a Huffman code for the 12 categories, where every codeword is at most 16 bits long
- Encode each DC residual as a binary string hsm where
  - h is the codeword of the residual’s category
  - s= sign of the residual; s=0 if negative, 1 if positive
  - m= the (k-1)-bit [(magnitude of the residual) -2^{k-1}]

Back to Top

7. Coding of the AC Terms (The AC Huffman

Table)

- Divide the range of the AC's into 10 categories
- Category k, k=1,2,...,10, consists of 2^k integers ranging from -2^k+1 to -2^{k-1}, and from 2^{k-1} to 2^k-1
- Describe each [run,level] by 8 bits 'NNNNSSSS'
  - NNNN = the value of run (i.e., the runlength)
  - SSSS = the category of level
  - If the runlength run>15, then
    - run=15p +r, r=0,1,...,15, $r=r_3r_2r_1r_0$ in binary
    - describe [run,level] by 11110000_1 11110000_2 ... 1111000_p $r_3r_2r_1r_0$SSSS

- Add an end-of-block EOB symbol after the last nonzero coefficient in each block

- The total number of symbols needed is 16*10+1+1=162

- Build a Huffman table for those 162 symbols, where every codeword is at most 16 bits long

- Encode each AC quantized term as hsm where
  - h is the codeword of the AC's [run,category]
  - s= sign of the term; s=0 if negative, 1 if positive
  - m= the (k-1)-bit [(magnitude of the term) -2^{k-1}]

Back to Top

8. Examples

- Take an 8×8 block of Lena B:

143	147	149	152	156	147	146	149
151	146	143	154	148	144	153	132
147	143	145	149	144	145	128	133
152	145	145	144	146	134	130	137
146	143	142	147	124	127	139	138
139	145	139	127	126	135	139	141
145	137	124	130	138	136	140	144
144	124	136	134	137	139	142	145

- Normalize B. It becomes NB=B-128:

15	19	21	24	28	19	18	21
23	18	15	26	20	16	25	4
19	15	17	21	16	17	0	5
24	17	17	16	18	6	2	9
18	15	14	19	-4	-1	11	10
11	17	11	-1	-2	7	11	13
17	9	-4	2	10	8	12	16
16	-4	8	6	9	11	14	17

- Perform DCT, resulting in a block D:

103.4	12.4	6.0	2.1	8.1	5.7	-0.7	-0.4
31.7	11.6	-22.8	-0.7	-0.1	-0.5	-4.7	-1.8
11.0	-21.2	-3.7	8.8	2.3	0.1	-0.2	5.6
0.2	-4.9	10.3	-8.6	-8.9	-2.5	-7.6	-1.4
4.9	-0.4	-1.9	-8.9	6.6	2.6	3.6	3.6
0.7	-0.9	-0.9	4.1	7.2	-15.5	2.8	1.8
-3.1	-1.0	-2.5	-5.5	-5.2	-6.7	10.7	-1.1
-2.9	-0.2	-1.2	-2.7	3.6	2.6	0.4	-6.4

- Quantize (round(D./Q)), resulting in Dq:

6	1	0	0	0	0	0	0
4	1	-2	0	0	0	0	0
1	-2	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

- zigzag ordering: 6 1 4 1 1 0 0 -2 -2 allzeros

Back to Top

9. Example Huffman Table for Lena

Zero Run	**Cetgory**	**Codelength**	**Codeword**
0	1	2	00
0	2	2	01
0	3	3	100
0	4	4	1011
0	5	5	11010
0	6	6	111000
0	7	7	1111000
.	.	.	.
1	1	4	1100
1	2	6	111001
1	3	7	1111001
1	4	9	111110110
.	.	.	.
2	1	5	11011
2	2	8	11111000
.	.	.	.
3	1	6	111010
3	2	9	111110111
.	.	.	.
4	1	6	111011
5	1	7	1111010
6	1	7	1111011
7	1	8	11111001
8	1	8	11111010
9	1	9	111111000
10	1	9	111111001
11	1	9	111111010
.	.	.	.
.	.	.	.
End of Block (EOB)		4	1010

Back to Top

10. Coding the Example Block

- zigzag ordering: 6 1 4 1 1 0 0 -2 -2 allzeros

- Huffman symbols: [run, category]
- Code:
- Bitrate (assuming 8 bits dor the DC residual symbol):

Back to Top

11. Decoding

1. Entropy-decode the bitstream back to the quantized blocks
2. Dequantize: multiply each block coefficient by the corresponding coefficient of the quantization matrix
3. Apply the inverse DCT transform on each block
4. Denormalize: add 128 to each coefficient

- Example: Reconstructed block B’

143	147	153	157	157	154	149	145
145	147	151	154	153	149	144	141
146	147	149	149	146	142	137	134
146	146	145	142	139	135	132	130
145	143	140	136	133	131	130	130
141	138	134	131	130	131	134	135
136	134	130	128	129	133	139	142
133	131	127	126	129	135	142	147

- Error block

0	0	-4	-5	-1	-7	-3	4
6	-1	-8	0	-5	-5	9	-9
1	-4	-4	0	-2	3	-9	-1
6	-1	0	2	7	-1	-2	7
1	0	2	11	-9	-4	9	8
-2	7	5	-4	-4	4	5	6
9	3	-6	2	9	3	1	2
11	-7	9	8	8	4	0	-2

- MSE=5.2

Back to Top

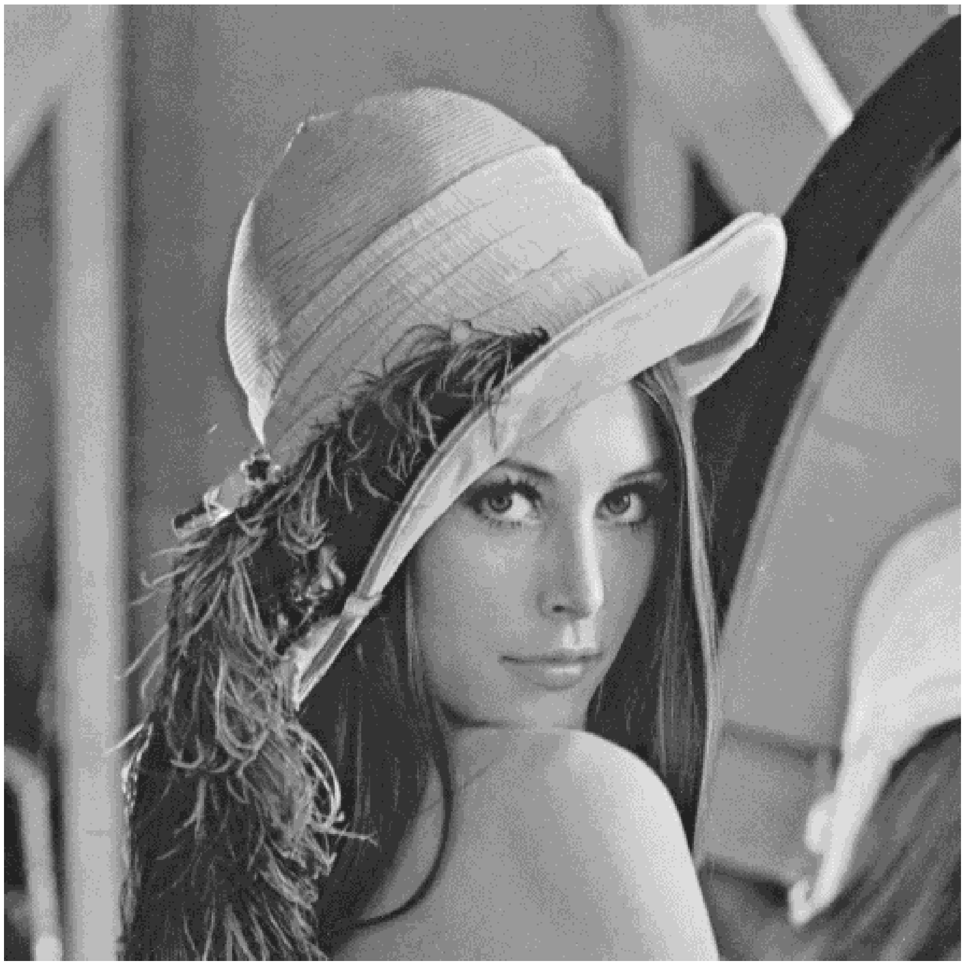
12. ## Extended JPEG

- Extended JPEG allows for several optional enhancements:
    - 12-bit/pixel input
    - Arithmetic coding is allowed as an alternative to Huffman coding
    - Adaptive quantization: Allows 5-bit scale change to the quantization matrix from one block to another
    - SPIFF: A file format that provides for the interchange of compressed image files between different application environments
    - Progressive transmission (PT)
        - Sequential PT
            - Spectral selection
            - Successive approximations
        - Hierarchical PT (pyramid encoding)
    - Tiling
    - Selective refinement

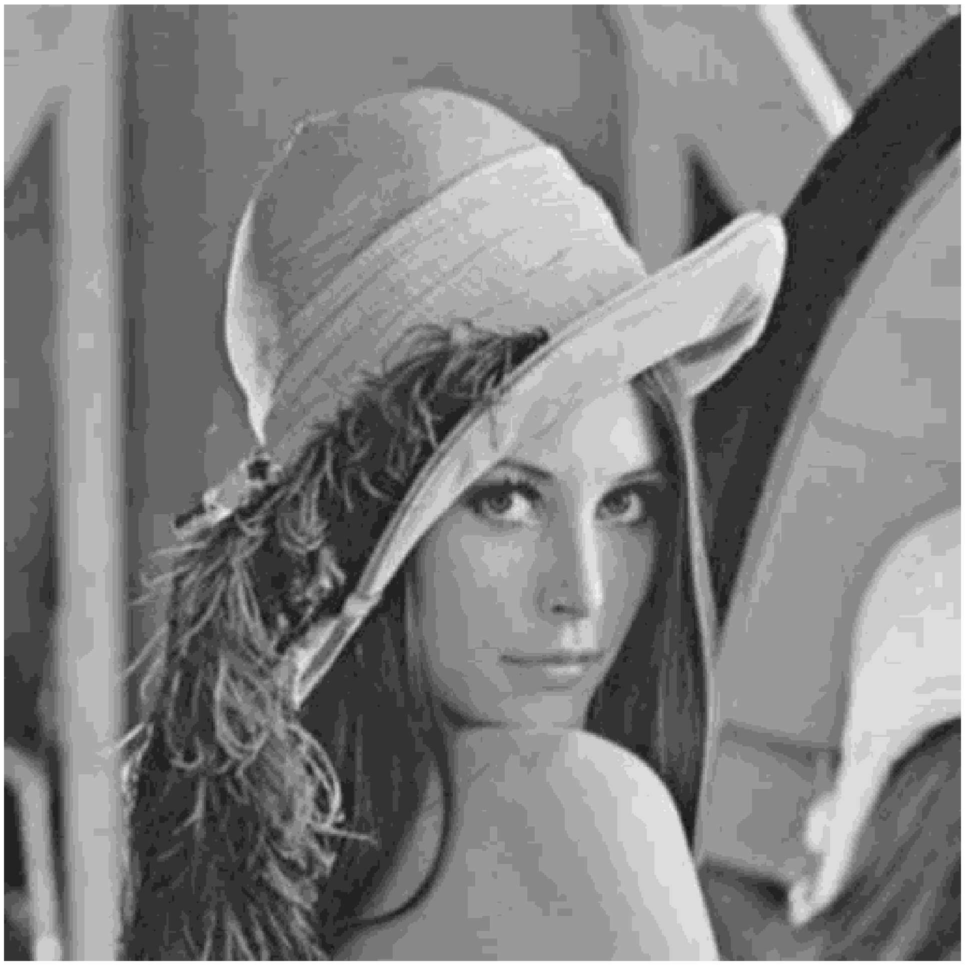
Back to Top

13. ## Performance of JPEG

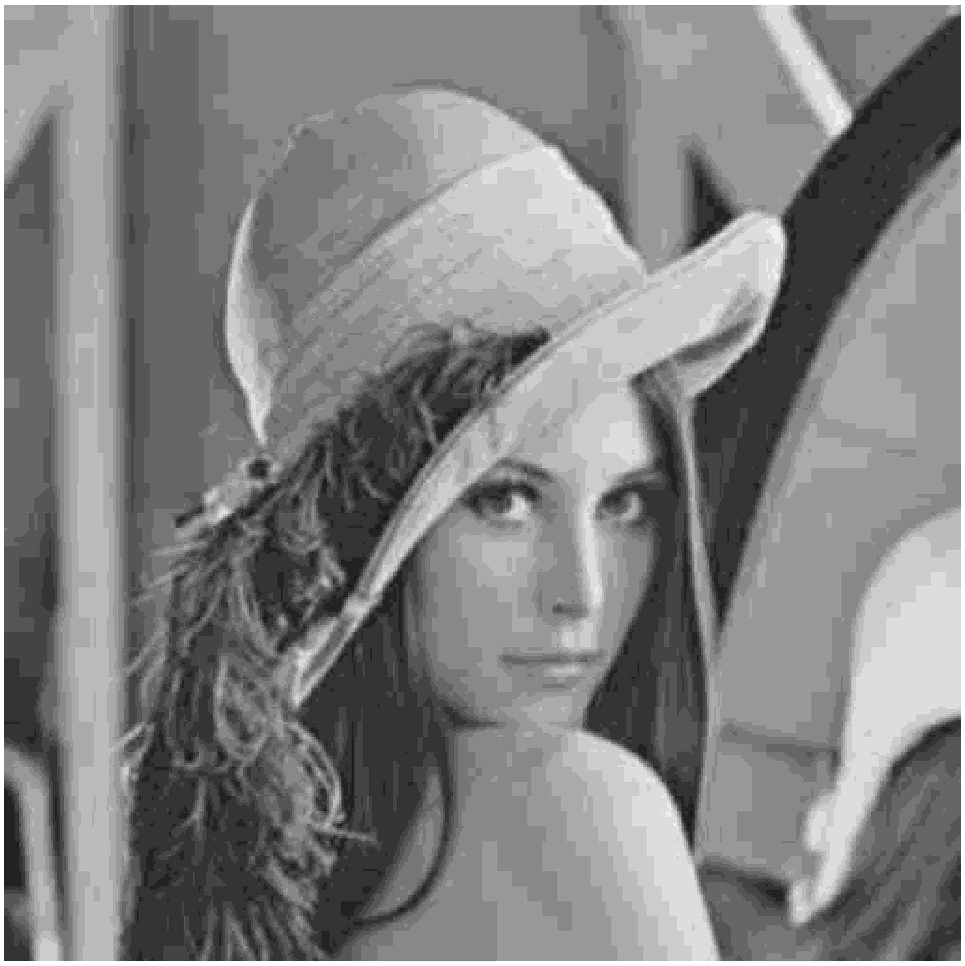
- Original Lena



- JPEG-compressed Lena at a compression ratio of 12:1



- JPEG-compressed Lena at a compression ratio of 20:1



- JPEG-compressed Lena at a compression ratio of 32:1

Back to Top

14. **MPEG (1 & 2): Basic Concepts**

- Video is a sequence of images called frames
- Color
  - Three components: red (R), green (G), and blue (B)
  - For compatibility with non-colored media, the RGB model was converted to an equivalent model --- YC_bC_r
    - Y is the luminance component, which was experimentally determined to be
      $$Y=0.299R+0.587G+0.114B$$
    - $C_b=B-Y$
    - $C_r=R-Y$
  - Y is referred to as **luma**, and C_b & C_r as **chroma**
- Every frame is really 3 images: one Y, one C_b and one C_r

- 8 bits/pixel for each of the three color components
- Because human vision is less sensitive to color, the C_b and C_r images are downsampled by 2 in each dimension (they are quarter the size of Y images)
- Much of MPEG processing is on the basis of a **macroblock**: A 16×16 luminance block with the two 8×8 associated chroma blocks

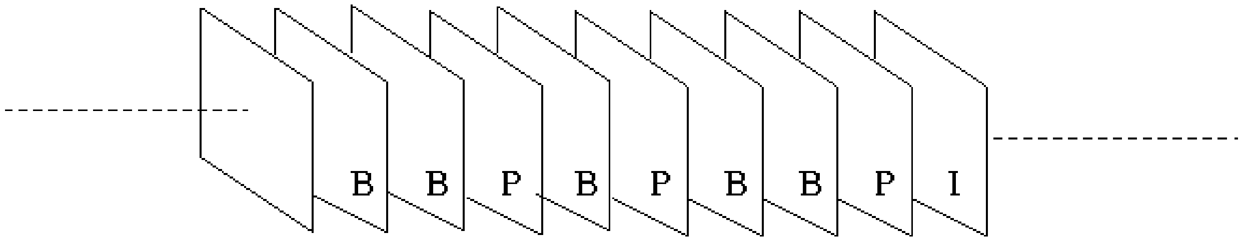
Back to Top

15. Modes of MPEG Compression

- Intraframe compression
  - Exploits spatial redundancy only
  - Operates on single frames independetly of other frames
- Interframe compression
  - Exploits both spatial and temporal redunadancies
  - Employs motion estimation (MS) without standardizing any MS algorithm
  - Derives motion-compensated predictions of frames
  - Finally, it performs JPEG-like compression on the residual frames

Back to Top

16. Types of Frames in MPEG



- MPEG has 3 types of frames: I, P, and B
- I frames are stricly intra compressed as in JPEG. Their purpose is to provide random access points to the video
- P frames are motion-compensated forward-predictive-coded frames; they are interframe compressed, and typically provide more compression than I frames
- B frames are motion-compensated bidirectionally-predictive-coded frames; they are interframe compressed, and typically provide the most compression
- The relative numbers of I, P and B frames are arbitrary

- An I frame must occur at least once every 132 frames to provide user-acceptible speed of random access to various parts of a video

Back to Top

17. Interframe Compression of P and B Blocks

- A motion-compensated prediction (i.e., approximation) f’ of a P/B frame f is made

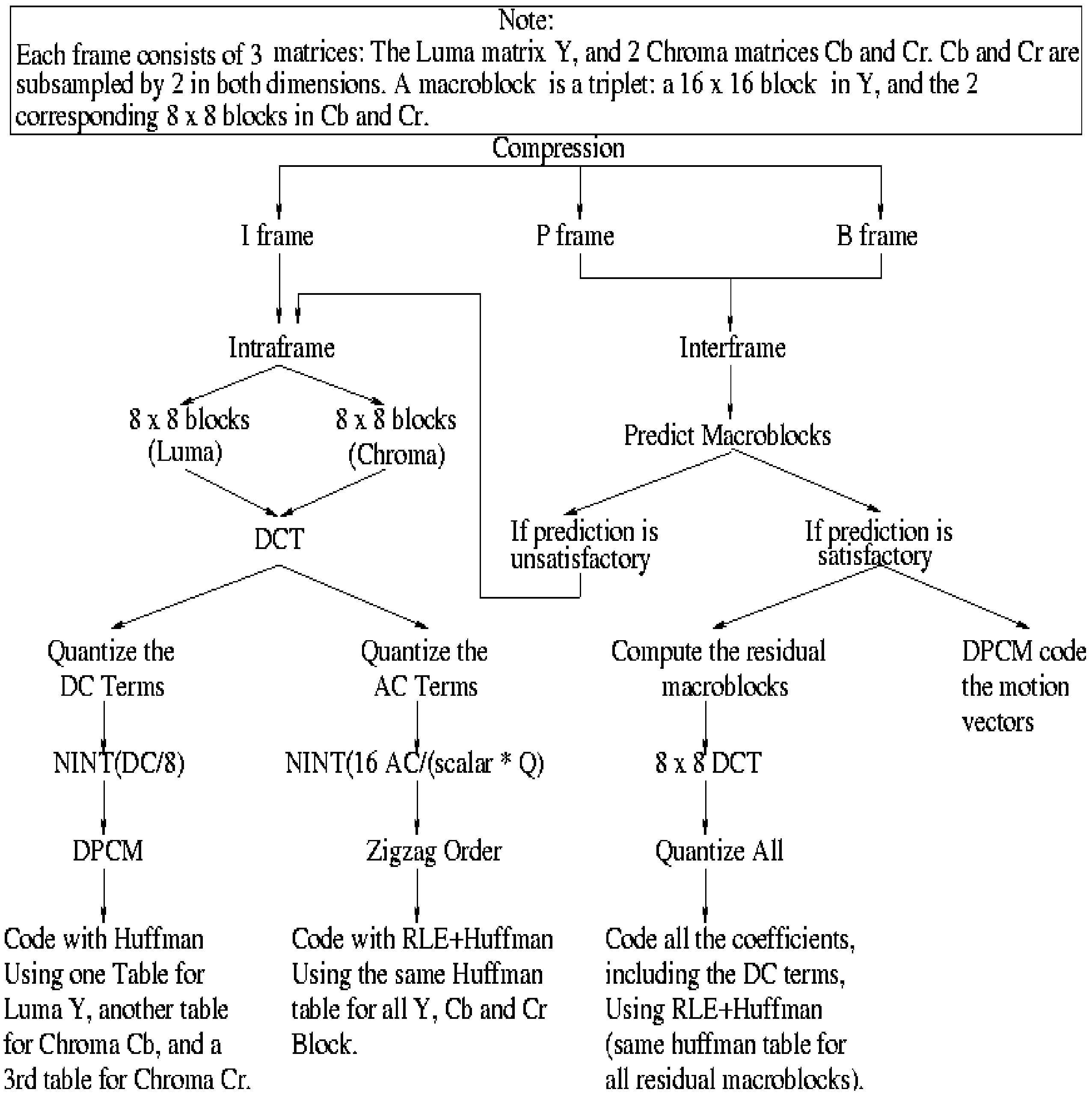
- The residual f-f’ is then compressed in a JPEG-like style

- Any macroblock of the original frame f may be strictly intra compressed if its prediction is deemed to be poor

- Issues to be addressed
  - Method of strict intra compression
  - Method of compressing residual macroblocks
  - Motion-compensated prediction

Back to Top

18. Flowchart of MPEG Compression



Note:
Each frame consists of 3 matrices: The Luma matrix Y, and 2 Chroma matrices Cb and Cr. Cb and Cr are subsampled by 2 in both dimensions. A macroblock is a triplet: a 16 x 16 block in Y, and the 2 corresponding 8 x 8 blocks in Cb and Cr.
Compression
I frame
P frame
B frame
Intraframe
Interframe
8 x 8 blocks (Luma)
8 x 8 blocks (Chroma)
Predict Macroblocks
DCT
If prediction is unsatisfactory
If prediction is satisfactory
Quantize the DC Terms
Quantize the AC Terms
Compute the residual macroblocks
DPCM code the motion vectors
NINT(DC/8)
NINT(16 AC/(scalar * Q)
8 x 8 DCT
DPCM
Zigzag Order
Quantize All
Code with Huffman Using one Table for Luma Y, another table for Chroma Cb, and a 3rd table for Chroma Cr.
Code with RLE+Huffman Using the same Huffman table for all Y, Cb and Cr Block.
Code all the coefficients, including the DC terms, Using RLE+Huffman (same huffman table for all residual macroblocks).

Back to Top

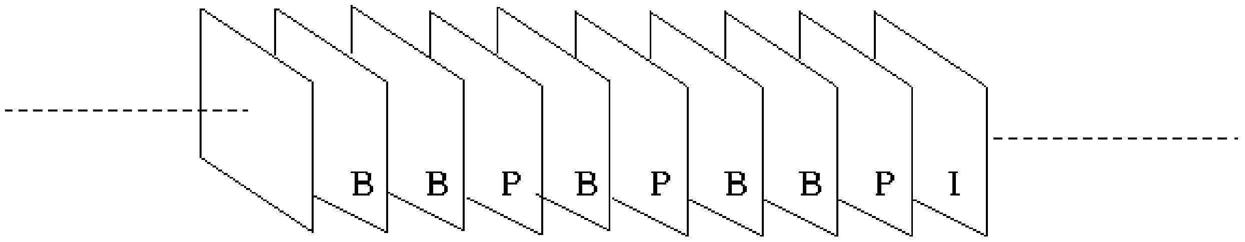
19. Motion-Estimation and Prediction

- Motion estimation is performed on the basis of macroblocks, using the 16×16 luminance blocks only
- Motion is assumed to be uniform across all the pixels of a macroblock
- Remark: There is a tradeoff in deciding the size of the basic block for motion estimation

- The block has to be sufficiently large to avoid ‘‘false hits’’
- The block has to be sufficiently small to avoid diverse motions within one single block
- The MPEG block size, 16×16, is a good compromise

Back to Top

20. Motion-Estimation and Prediction for P Frames

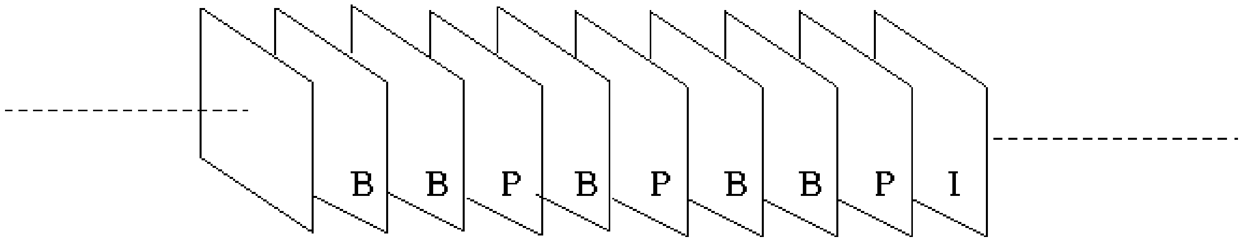


- Consider a P-frame P
- P will be predicted (i.e., approximated) from one single reference frame R
- R is the most recent (decoded) I or P frame
- For each macroblock MB of P, find the closest matching macroblock MB’ in the reference frame R
- If the MB-to-MB’ match is satisfactory, then
  - treat MB’ as the prediction (i.e., approximation) of MB
  - record the motion (i.e., displacement) vector between the two macroblocks (allowing half-pixel accuracy)
  - Compute and compress the macroblock residual MB-MB’ (luma and chroma)
- If the MB-to-MB’ match is found to be unsatisfactory, then the macroblock MB is strictly intra compressed as is done in I frames
- The motion vectors of all the macroblocks of P exhibit redundancy due to similar (or sometimes identical) motion experienced by many neighboring macroblocks
- This redundancy is exploited by coding the consecutive differential values of motion vectors (i.e., DPCM)
- Remark 1: MPEG does not standardize the decision mechanism for judging whether or or not a match between two macroblocks is satisfactory
- Remark 2: A typical decision mechanism involves computing an error measure between the luminance of the two macro blocks. The match is treated as satisfory if and only if the error

is below a certain threshold. Possible error measures include mean-square error (MSE), mean absolute-difference error (MAD), and variance(MB-MB')/variance(MB).

Back to Top

21. Motion-Estimation and Prediction for B Frames



- Consider a B-frame B
- B will be predicted (i.e., approximated) from TWO reference frames R_1 and R_2
- R_1 is the most recent (decoded) past I/P frame, and R_2 is the nearest (decoded) future I/P frame
- For each macroblock MB of B, find the closest matching macroblock MB_1 in the reference frame R_1, and the closest matching macroblock MB_2 in R_2
- The predicted macroblock is PM=NINT($alpha_1 MB_1 + alpha_2 MB_2$)
  - $alpha_1$=0.5 and $alpha_2$=0.5 if both matches are satisfactory
  - $alpha_1$=1 and $alpha_2$=0 if only the 1st match is satisfactory
  - $alpha_1$=0 and $alpha_2$=1 if only the 2nd match is satisfactory
  - $alpha_1$=0 and $alpha_2$=0 if neither match is satisfactory
- Compute and compress the macroblock residual MB-PM (luma and chroma)
- If neither match is found to be satisfactory, then the macroblock MB is strictly intra compressed as is done in I frames
- Record the motion vector(s) between the MB and the other one or two macroblocks (allowing half-pixel accuracy)
- Again, the motion-vector redundancy is exploited by coding the consecutive differential values of motion vectors (i.e., DPCM)
- Remark: The prediction mode chosen is 2-bit coded and passed on along with the

macroblock header information

Back to Top

22. MPEG2

- Higher data rates than MPEG1

- MPEG2 allows for higher quality source images
  - 4:2:0 (chroma subsamples only horizontally)
  - 4:4:4 (no subsampling of chroma)
  - Note: 4:2:2 is what's supported by MPEG1

- MPEG2 allows for finer quantization and for specifying separate quantization table for luma and chroma

- MPEG2 allows for finer adjustment of quantization scale factor, used in intra compression

- MPEG2 allows for interlaced video

- MPEG2 supports error concealment (of lost macroblocks)

- MPEG2 supports scalable compression
  - SNR-scalability: by sending bands of DCT coefficients
  - spatial-scalability: pixel resolution by down-/up-sampling
  - temporal-scalability: different frame rates by skipping frames

- MPEG2 has a **Profile** and **Level** structure
  - Profiles are algorithmic elements included in MPEG2
  - Levels are upper bounds on parameter values

- Profiles (profiles are backward-compatible)
  1. Simple: No use of B frames
  2. Main: what was described earlier, but no scalability
  3. SNR-scalable
  4. Spatially scalable
  5. High: temporally scalable, higher-quality source data (4:2:0, 4:2:2 and possibly 4:4:4 in the future)

- Levels
  1. Low: 352 × 240 frame size
  2. Main: 720 × 480 frame size
  3. High 1440: 1440 × 1152 frame size
  4. Very High: 1920 × 1080 frame size

Back to Top

23. References

1. B. pennebaker and J. L. Mitchell, JPEG Still Image Data Compression Standard, Van Nostrand reinhold, New York 1993.
2. ISO-11172-2: Generic Coding of moving pictures and associated audio (MPEG-1)
3. ISO-13818-2: Generic Coding of moving pictures and associated audio (MPEG-2)

Back to Top

24. Links to other Standards

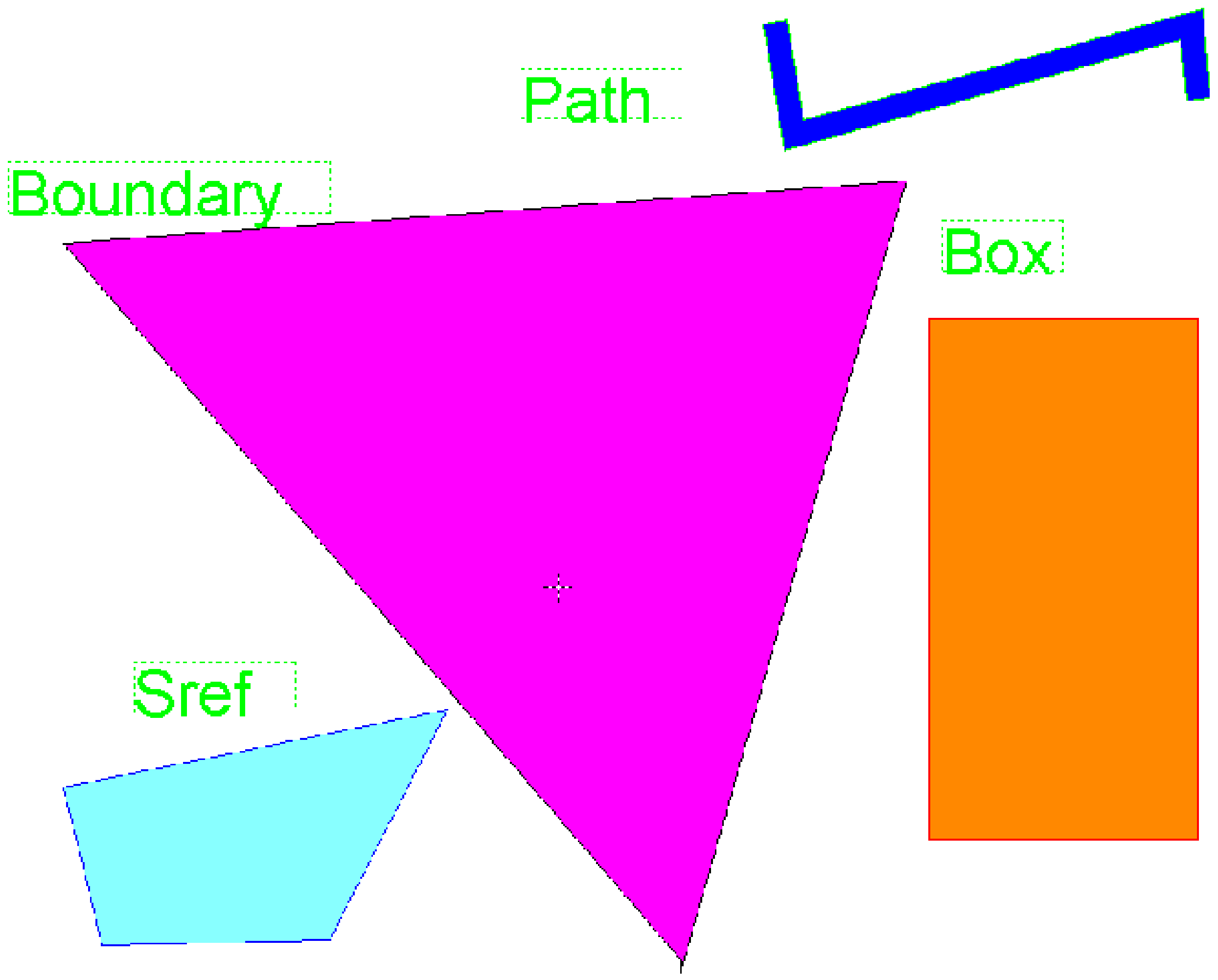
- JPEG 2000
- An overview of MPEG 4
- MPEG 2/MPEG 4 Audio Coding: Advanced Audio Coding (AAC)
- MPEG 1 Layer 3 Audio (MP3)
- Digital Audio Compression (AC-3) Standard and its 2001 Revision A

Back to Top

GDSII format

INDEX

example



1. text presentation of GDSII file in KEYformat
2. hex presentation of same file
3. GDSII file

introduction

GDSII Stream format is the standard file format for transfering/archiving 2D graphical design data. It contains a hiearchy of structures, each structure containing elements (boundary/polygon, path/polyline, text,box, structure references, structure array references). The elements are situated on layers. It is a binary format that is platform independent, because it uses internally defined formats for its data types. While reading GDSII files, the GDSII internal data types (like reals, integers etc.) need to be converted to the platform/CAE package datatypes that are used.The GDSII format is a sequential list of records, each record contains a header to tell what information is in the record.The order of the record needs to be according to the GDSII BNF, because of this strict organization it is relativly easy to parse. The maximum number of vertixes is officially only 200 x,y pairs, but many packages can read up to the absolute maximum of 64k/2=32k, simple because this is the maximum record lenght that can be specified (two bytes).The format is hard to read, since it is binary, for that viewers are available to view (boolean) the contents as ASCII. Also an ASCII format has been developed (KEY format) which is more than just a text representation. It is possible to convert GDSIIformat to KEYformat and back. KEYformat has extended the basic primitives to contain cicrles, arcs, polygons/polylines with arc segments.

Bachus Nauer Forms

The Bachus Nauer Form uses the following symbols:

Symbol Name	Symbol	Meaning
Double Colon	::	"Is composed of."
Square brackets	[ ]	An element which can occor zero or one time.
Braces	{ }	Choose one of the elements within the braces.
Braces with an asteriks	{ }*	The elements within the braces can occur zero or more times.
Braces with a plus	{ }+	The elements within braces must occur one or more times.
Angle brackets	< >	These elements are further defined as a seperate entitie in the syntax list.
Vertical bar	|	Or

GDSII BNF

The following is the Bachus Naur Form of the GDSI format, the words in capital are the names of *RECORDS*

<stream format>	::=	HEADER BGNLIB LIBNAME [REFLIBS] [FONTS] [ATTRTABLE] [GENERATIONS] [<FormatType>] UNITS {<structure>}* ENDLIB
<FormatType>	::=	FORMAT | FORMAT {MASK}+ ENDMASKS
<structure>	::=	BGNSTR STRNAME [STRCLASS] {<element>}* ENDSTR
<element>	::=	{<boundary> | <path> | <sref> | <aref> | <text> | <node> | <box>} {<property>}* ENDEL
<boundary>	::=	BOUNDARY [ELFLAGS] [PLEX] LAYER DATATYPE XY
<path>	::=	PATH [ELFLAGS] [PLEX] LAYER DATATYPE [PATHTYPE][WIDTH] XY
<sref>	::=	SREF [ELFLAGS] [PLEX] SNAME [<strans>] XY
<aref>	::=	AREF [ELFLAGS] [PLEX] SNAME [<strans>] COLROW XY
<text>	::=	TEXT [ELFLAGS] [PLEX] LAYER <textbody>
<node>	::=	NODE [ELFLAGS]. [PLEX] LAYER NODETYPE XY
<box>	::=	BOX [ELFLAGS] [PLEX] LAYER BOXTYPE XY
<textbody>	::=	TEXTYPE [PRESENTATION] [PATHTYPE] [WIDTH] [<strans>] XY STRING
<strans>	::=	STRANS [MAG] [ANGLE]
<property>	::=	PROPATTR PROPVALUE

Record header

The Stream format output file is composed of variable length records. Record length is measured in bytes. The minimum record length is four bytes. Within the record, two bytes (16 bits) is a word. The 16 bits in a word are numbered 0 to 15, left to right.The first four bytes of a record compose the recordheader. The first two bytes of the recordheader contain a count (in eight-bit bytes) of the total record length, so the maximum length is 65536 (64k). The next record starts immediately after the last byte of the previous record.The third byte of the header is the record type. The fourth byte of the header identifies the type of data contained within the record. The fifth until count bytes of a record contain the data.

Bitnr	0	1	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Word1	Total Record length in bytes															
Word2	*Record Type*								*Data type*							
Word3	Data until Word n (total record length/2)															

Data Types

The fourth byte in the record header contains the data type for the rest of the record. The record length is used to find the number of items of the specified datatype.

Data Type	Value
No Data	0
Bit Array	1
Two Byte Signed Integer	2
Four Byte Signed Integer	3
Four Byte Real	4 (not used)
Eight Byte Real	5
ASCII string	6

1. **Bit Array:**
   A bit array is a word which uses the value of a particular bit or group of bits to represent data. A bit array allows oneword to represent a number of simple pieces of information.

2. **Two-Byte Signed Integer:**
   2-byte integer = 1 word 2s-complement representation. The range of two-byte signed integers is -32,768 to 32,767.

   The following is a representation of a two-byte integer, where S is the sign and M is the magnitude.

   smmmmmmm mmmmmmmm

   The following are examples of two-byte integers:

   00000000 00000001 = 1
   00000000 00000010 = 2
   00000000 10001001 = 137
   11111111 11111111 = -1
   11111111 11111110 = -2
   11111111 01110111 = -137

3. **Four-Byte Signed Integer:**
   4-byte integer = 2 word 2s-complement representation

   The range of four-byte signed integers is -2,147,483,648 to 2,147,483,647.

   The following is a representation of a four-byte integer, where S is the sign and M is the magnitude.

   smmmmmmm mmmmmmmm mmmmmmmm mmmmmmmm

   The following are examples of four-byte integers:

   00000000 00000000 00000000 00000001 = 1

00000000 00000000 00000000 00000010 = 2
00000000 00000000 00000000 10001001 = 137
11111111 11111111 11111111 11111111 = -1
11111111 11111111 11111111 11111110 = -2
11111111 11111111 11111111 01110111 = -137

4. **Four-Byte Real**
   4-byte real = 2-word floating point representation

   (See 5.)

5. **Eight-Byte Real**
   8-byte real = 4-word floating point representation

   For all non-zero values:

   - A floating point number has three parts: the sign, the exponent, and the mantissa.
   - The value of a floating point number is defined as:
   - (Mantissa) x (16 raised to the true value of the exponent field).
   - The exponent field (bits 1-7) is in Excess-64 representation.
   - The 7-bit field shows a number that is 64 greater than the actual exponent.
   - The mantissa is always a positive fraction >=1/16 and <1. For a 4-byte real, the mantissa is bits 8-31. For an 8-byte real, the mantissa is bits 8-63.
   - The binary point is just to the left of bit 8.
   - Bit 8 represents the value 1/2, bit 9 represents 1/4, etc.
   - In order to keep the mantissa in the range of 1/16 to 1, the results of floating point arithmetic are normalized. Normalization is a process where by the mantissa is shifted left one hex digit at a time until its left FOUR bits represent a non-zero quantity. For every hex digit shifted, the exponent is decreased by one. Since the mantissa is shifted four bits at a time, it is possible for the left three bits of the normalized mantissa to be zero. A zero value, also called true zero, is represented by a number with all bits zero.

The following are representations of 4-byte and 8-byte reals, where S is the sign, E is the exponent, and M is the magnitude. Examples of 4-byte reals are included in the following pages, but 4-byte reals are not used currently. The representation of the negative values of real numbers is exactly the same as the positive, except that the highest order bit is 1, not 0. In the eight-byte real representation, the first four bytes are exactly the same as in the four-byte real representation. The last four bytes contain additional binary places for more resolution.

4-byte real:

SEEEEEEE MMMMMMMM MMMMMMMM MMMMMMMM

8-byte real:

SEEEEEEE MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM

Examples of 4-byte real:

Note: In the first six lines of the following example, the 7-bit exponent field = 65. The actual exponent is 65-64=1.

01000001 00010000 00000000 00000000 = 1
01000001 00100000 00000000 00000000 = 2
01000001 00110000 00000000 00000000 = 3
11000001 00010000 00000000 00000000 = -1
11000001 00100000 00000000 00000000 = -2
11000001 00110000 00000000 00000000 = -3
01000000 10000000 00000000 0000000 = 0 .5
01000000 10011001 10011001 1001100 = 1 .6
01000000 10110011 00110011 0011001 = 1 .7
01000001 00011000 00000000 00000000 = 1.5
01000001 00011001 10011001 10011001 = 1.6
01000001 00011011 00110011 00110011 = 1.7
00000000 00000000 00000000 00000000 = 0
01000001 00010000 00000000 00000000 = 1
01000001 10100000 00000000 00000000 = 10
01000010 01100100 00000000 00000000 = 100
01000011 00111110 00000001 00000000 = 1000
01000100 00100111 00010000 00000000 = 10000
01000101 00011000 01101010 00000000 = 100000

6. **ASCII String**
A collection of ASCII characters, where each character is represented by one byte. All odd length strings must be padded with a null character (the number zero), and the byte count for the record containing the ASCII string must include this null character. Stream read-in programs must look for the null character and decrease the length of the string by one if the null character is present.

Record Types Overview

The following table gives an overview of all the record that are used within a GDSII file.

Nr.	Code	Mnemonic	Data Type	description
0	0002	HEADER	Two-Byte Signed Integer	version number
1	0102	BGNLIB	Two-Byte Signed Integer	begin of library, last modification date and time
2	0206	LIBNAME	Two-Byte Signed Integer	name of library
3	0305	UNITS	Eight-Byte Real	user and database units
4	0400	ENDLIB	No Data	end of library
5	0502	BGNSTR	Two-Byte Signed Integer	begin of structure + creation and modification time
6	0606	STRNAME	ASCII string	name of structure

7	0700	ENDSTR	No Data	end of structure
8	0800	BOUNDARY	No Data	begin of boundary element
9	0900	PATH	No Data	begin of path element
10	0A00	SREF	No Data	begin of structure reference element
11	0B00	AREF	No Data	begin of array reference element
12	0C00	TEXT	No Data	begin of text element
13	0D02	LAYER	Two-Byte Signed Integer	layer number of element
14	0E02	DATATYPE	Two-Byte Signed Integer	Datatype number of element
15	0F03	WIDTH	Four-Byte Signed Integer	width of element in db units
16	1003	XY	Four-Byte Signed Integer	list of xy coordinates in db units
17	1100	ENDEL	No Data	end of element
18	1206	SNAME	ASCII string	name of structure reference
19	1302	COLROW	Two-Byte Signed Integer	number of colomns and rows in array reference
21	1500	NODE	No Data	begin of node element
22	1602	TEXTTYPE	Two-Byte Signed Integer	texttype number
23	1701	PRESENTATION	Bit Array	text presentation, font
25	1906	STRING	ASCII string	character string for text element
26	1A01	STRANS	Bit Array	array reference, structure reference and text transform flags
27	1B05	MAG	Eight Byte Real	magnification factor for text and references
28	1C05	ANGLE	Eight Byte Real	rotation angle for text and references
31	1F06	REFLIBS	ASCII string	name of referenced libraries
32	2006	FONTS	ASCII string	name of text fonts definition files
33	2102	PATHTYPE	Two-Byte Signed Integer	type of PATH element end (rounded, square)
34	2202	GENERATIONS	Two-Byte Signed Integer	number of deleted structure ?????
35	2306	ATTRTABLE	ASCII string	attribute table, used in combination with element properties
38	2601	ELFLAGS	Two-Byte Signed Integer	template data
42	2A02	NODETYPE	Two-Byte Signed Integer	node type number for NODE element

43	2B02	PROPATTR	Two-Byte Signed Integer	attribute number
44	2C06	PROPVALUE	ASCII string	attribute name
45	2D00	BOX	No Data	begin of box element
46	2E02	BOXTYPE	Two-Byte Signed Integer	boxtype for box element
47	2F03	PLEX	Four-Byte Signed Integer	plex number
50	3202	TAPENUM	Two-Byte Signed Integer	Tape Number
51	3302	TAPECODE	Two-Byte Signed Integer	Tape code
54	3602	FORMAT	Two-Byte Signed Integer	format type
55	3706	MASK	ASCII string	list of layers
56	3800	ENDMASKS	No Data	end of MASK

Record types description

Records are always an even number of bytes long. The first four bytes of a record are the *record header*. If a record contains ASCII string data and the ASCII string is an odd number of bytes long, the data is padded with a null character. This paragraph lists the record types with a brief description of each. The descriptions include the record name and a four-digit number in brackets. The first two numbers within the brackets are the record type, and the last two numbers in brackets are the data type. All record numbers are expressed in hexadecimal.

0	HEADER	0002	Two-Byte Signed Integer

Contains two bytes of data representing the Stream version number.

1	BGNLIB	0102	Two-Byte Signed Integer

Contains the last modification time of a library (two bytes each for year, month, day, hour, minute, and second), the time of last access (same format), and marks the beginning of a library.

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
word1	l C (hex) # of bytes in record															
word2	01 (hex) 02 (hex)															
word3	year (lastmodification time)															
word4	month															
word5	day															
word6	hour															
word7	minute															
word8	second															
word9	year (last access time)															
word10	month															
word11	day															
word12	hour															
word13	minute															
word14	second															

2	LIBNAME	0206	ASCII String

Contains a string which is the library name. The library name must follow UNIX filename conventions for length and valid characters. The library name may include the file extension (.sf or db in most cases).

3	UNITS	0305	Eight-Byte Real

Contains two eight-byte real numbers. The first number is the size of a database unit in user units. The second number is the size of a database unit in meters. For example, if you create a library with the default units (user unit = 1 micron and 1000 database units per user unit), the first number is .001, and the second number is 1E-9. Typically, the first number is less than 1, since you use more than 1 database unit per user unit. To calculate the size of a user unit in meters, divide the second number by the first.

4	ENDLIB	0400	No Data

Marks the end of a library.

5	BGNSTR	0502	Two-Byte Signed Integer

Contains the creation time and last modification time of a structure (in the same format as the BGNLIB record), and marks the beginning of a structure.

6	STRNAME	0606	ASCII String

Contains a string which is the structure name. A structurename may be up to 32 characters long. Legal structurename characters are:

- A through Z
- a through z
- 0 through 9
- Underscore (_)
- Question mark (?)
- Dollar sign ($)

7	ENDSTR	0700	No Data

Marks the end of a structure.

8	BOUNDARY	0800	No Data

Marks the beginning of a boundary element.

9	PATH	0900	No Data

Marks the beginning of a path element.

10	SREF	0A00	No Data

Marks the beginning of an Sref (structure reference) element.

11	AREF	0B00	No Data

Marks the beginning of an Aref (array reference) element.

12	TEXT	0C00	No Data

Marks the beginning of a text element.

13	LAYER	0D02	Two-Byte Signed Integer

Contains two bytes which specify the layer. The value of the layer must be in the range of 0 to 255.

14	DATATYPE	OEO2	Two-Byte Signed Integer

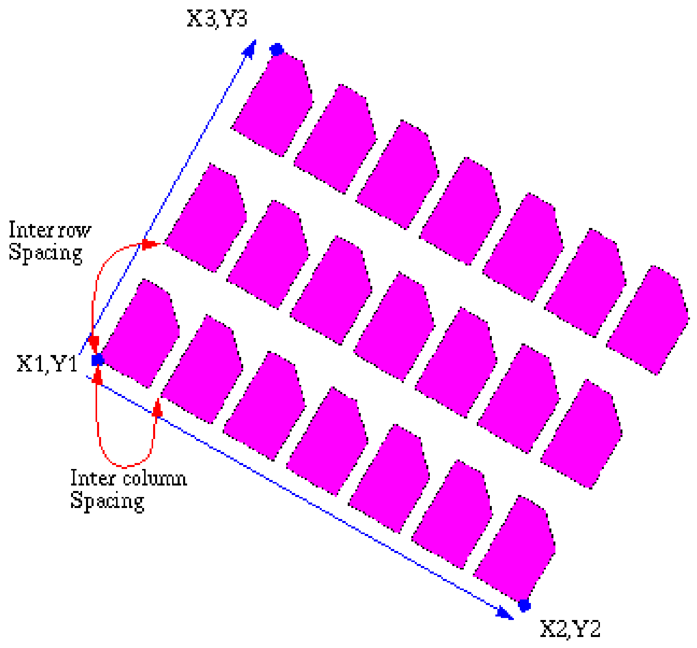
Contains two bytes which specify the datatype. The value of the datatype must be in the range of 0 to 255.

15	WIDTH	0F03	Two-Byte Signed Integer

Contains four bytes which specify the width of a path or text lines in database units. A negative value for width means that the width is absolute, that is, the width is not affected by the magnification factor of any parent reference. If omitted, zero is assumed.

16	XY	1003	Two-Byte Signed Integer

- Contains an array of XY coordinates in database units. Each X or Y coordinate is four bytes long. Path elements may have a minimum of 2 and a maximum of 200 coordinates. Boundary and border elements may have a minimum of 4 and a maximum of 200 coordinates. The first and last coordinates of a boundary or border must coincide.
- A text, or Sref element may have only one coordinate.
- An Aref has exactly three coordinates. In an Aref, the first coordinate is the array reference point (origin point). The other two coordinates are already rotated, reflected as specified in the STRANS record (if specified). So in order to calculate the intercolomn and interrow spacing, the coordinates must be mapped back to their original position, or the vector lenght (x1,y1-> x3,y3) must be divided by the number of row etc. . The second coordinate locates a position which is displaced from the reference point by the inter-column spacing times the number of columns. The third coordinate locates a position which is displaced from the reference point by the inter-row spacing times the number of rows. For an example of an array lattice see the next picture.



Aref rotated -30 degrees.

- A node may have from one to 50 coordinates.
- A box must have five coordinates, with the first and last coordinates being the same.

17	ENDEL	1100	No Data

Marks the end of an element.

18	SNAME	1206	ASCII string

Contains the name of a referenced structure.See also STRNAME.

19	COLROW	1302	Two-Byte Signed Integer

Contains four bytes. The first two bytes contain the number of columns in the array. The third and fourth bytes contain the number of rows. Neither the number of columns nor the number of rows may exceed

32,767 (decimal), and both are positive. See also AREF.

21	NODE	1500	No Data

Present Marks the beginning of a node

22	TEXTTYPE	1602	Two-Byte Signed Integer

Contains two bytes representing texttype. The value of the texttype must be in the range 0 to 255.

23	PRESENTATION	1701	Bit Array

Contains one word (two bytes) of bit flags for text presentation. Bits 10 and 11, taken together as a binary number, specify the font (00 means font 0, 01 rneans font 1, 10 means font 2, and 11 means font 3). Bits 12 and 13 specify the vertical justification (00 means top, 01 means middle, and 10 means bottom). Bits 14 and 15 specify the horizontal justification (00 means left, 01 means center, and 10 means right). Bits 0 through 9 are reserved for future use and must be cleared. If this record is omitted, then top-left justification and font 0 are assumed. The following shows a PRESENTATION record.

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
word1	6 (hex) # of bytes in record															
word2	17 (hex)								01 (hex)							
word3	unused										font number		vertical presentaion		horizontal presentation	

25	STRING	1906	ASCII String

Contains a character string, up to 512 characters long, for text presentation.

26	STRANS	1A01	Bit Array

Contains two bytes of bit flags for Sref, Aref, and text transfornation. Bit 0 (the leftmost bit) specifies reflection. If bit 0 is set, the element is reflected about the X-axis before angular rotation. For an Aref, the entire array is reflected, with the individual array members rigidly attached. Bit 13 flags absolute magnification. Bit 14 flags absolute angle. Bit 15 (the rightmost bit) and all remaining bits are reserved for future use and must be cleared. If this record is omitted, the element is assumed to have no reflection, non-absolute magnification, and non- absolute angle.
The following shows a STRANS record.

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
word1	6 (hex) # of bytes in record															
word2	1A (hex)								01 (hex)							
word3	reflection	unused												absolute magnification	absolute angle	unused

27	MAG	1B05	Eight Byte Real

Eight-Byte Real Contains a double-precision real number (8 bytes), which is the magnification factor. If this record is omitted, a magnification factor of one is assumed.

28	ANGLE	1C05	Eight Byte Real

Eight-Byte Real Contains a double-precision real number (8 bytes), which is the angular rotation factor. The angle of rotation is measured in degrees and in the counterclockwise direction. For an Aref, the ANGLE rotates the entire array (with the individual array members rigidly attached) about the array reference point. For COLROW record information, the angle of rotation is already inlcuded in the coordinates. If this record is omitted, an angle of zero degrees is assumed.

31	REFLIBS	1F06	ASCII String

Contains the names of the reference libraries. This record must be present if any reference libraries are bound to the working library. The name of the first reference library starts at byte 5 (immediately following the record header) and continues for 44 bytes. The next 44 bytes contain the name of the second library. The record is extended by 44 bytes for each additional library (up to 15) which is bound for reference. The reference library names may include directory specifiers (separated with "/") and an extension (separated with "."). If either the first or second library is not named, its place is filled with nulls.

32	FONTS	2006	ASCII String

Contains the names of the textfont definition files. This record must be present if any of the four fonts have acorresponding textfont definition file. This record must not be present if none of the fonts have a textfont definition file. The textfont filename of font 0 starts the record, followed by the textfont files of the remaining three fonts.Each filename is 44 bytes long. The filename is padded withnulls if the name is shorter than 44 bytes. The filename is null if no textfont definition corresponds to the font. The textfont filenames may include directory specifiers (separated with "/" and an extension (separated with ".").

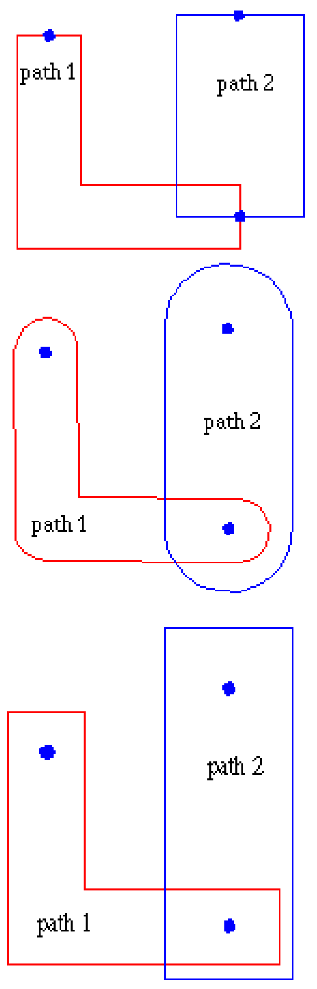
33	PATHTYPE	2102	Two-Byte Signed Integer

This record contains a value that describes the type of path endpoints. The value is

- 0 for square-ended paths that endflush with their endpoints
- 1 for round-ended paths
- 2 for square-ended paths that extend a half-width beyond their endpoints

If not specified, a Path-type of 0 is assumed.

The following picture shows the pathtypes

	Pathtype 0 produces a square-ended path, ending flush with thedigitized endpoints. This is the de-fault pathtype if none is specified
	Pathtype 1 produces a round-ended path. The two ends aresemicircular with center at thedigitized endpoints.
	Pathtype 2 produces a square-ended path. The ends of the pathextend beyond the digitized end-points by one-half the path width.

34	GENERATIONS	2202	Two-Byte Signed Integer

This record contains a value to indicate the number of copies of deleted or back-up structures to retain. This numbermust be at least 2 and not more than 99. If the GENERATION record is omitted, a value of 3 is assumed.

35	ATTRTABLE	2306	Two-Byte Signed Integer

Contains the name of the attribute definition file. This record is present only if an attribute definition file is bound to the library. The attribute defenition filename may include directory specifiers (separated with "/") and an extension (separated with "."). Maximum record size is 44 bytes.

36	STYPTABLE	2502	Two-Byte Signed Integer

Unrelesed Feature

37	STRTYPE	2502	Two-Byte Signed Integer

Unrelesed Feature

38	ELFLAGS	2601	Bit Array

Contains two bytes of bit flags. Bit 15 (the rightmost bit)specifies Template data. Bit 14 specifies External data(also referred to as Exterior data). All other bits are currently unused and must be cleared to 0. If this record isomitted, all bits are assumed to be 0. The following shows an ELFLAGS record.

For additional information on Template data, consult the GDSII Reference Manual. For additional information on External data, consult the CustomPlus User's Manual.

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
word1	6 (hex) # of bytes in record															
word2	26 (hex)								01 (hex)							
word3	unused														external data	template data

39	ELKEY	2703	Two-Byte Signed Integer

(Unreleased feature)

40	LINKTYPE	28	Two-Byte Signed Integer

(Unreleased feature)

41	LINKKEYS	29	Two-Byte Signed Integer

(Unreleased feature)

42	NODETYPE	2A02	Two-Byte Signed Integer

Contains two bytes which specify nodetype. The value ofthe nodetype must be in the range of 0 to 255.

43	PROPATTR	2B02	Two-Byte Signed Integer

Contains two bytes which specify the attribute number. The attribute number is an integer from 1 to 127. Attribute numbers 126 and 127 are reserved for the user integer and userstring (CSD) properties which existed prior to Release 3.0.

44	PROPVALUE	2C06	ASCII string

Contains the string value associated with the attribute named in the preceding PROPATTR record. Maximumlength is 126 characters. The attribute-value pairs associated with any one element must all have distinct attribute numbers. Also, the total amount of property data that may be associated with any one element is limited: thetotal length of all the strings, plus twice the number of attribute-value pairs, must not exceed 128 (or 512 if the element is an Sref, Aref, contact, nodeport, or node).

For example, if a boundary element uses property attribute2 with property value "metal," and property attribute 10 with property value "property," the total amount of property data is 18 bytes. This is 6 bytes for "metal" (odd-length strings must be padded with a null) + 8 for "property" + 2 times the 2 attributes (4) = 18.

45	BOX	2D00	No Data

Marks the beginning of a box element.

46	BOXTYPE	2E02	Two-Byte Signed Integer

Contains two bytes which specify boxtype. The value of the boxtype must be in the range of 0 to 255.

47	PLEX	2F03	Two-Byte Signed Integer

A unique positive number which is common to all elementsof the plex to which this element belongs. The head of the plex is flagged by setting the seventh bit; therefore, plexnumbers should be small enough to occupy only the right-most 24 bits. If this record is not present, the element is not a plex member Applies to Pathtype 4.Contains four bytes which specify indatabase units the distance a path outline begins before orafter the last point of the path. Value can be negative.

50	TAPENUM	3202	Two-Byte Signed Integer

Contains two bytes which specify the number of the current reel of tape for a multi-reel Stream file. For the first tape, the TAPENUM is 1: for the second tape, the TAPENUM is 2. For each additional tape, increment the TAPENum by one.

51	TAPECODE	3302	Two-Byte Signed Integer

Contains 12 bytes. This is a unique 6-integer code which iscommon to all the reels of multi-reel Stream file. It verifies that the correct reels are being read.

52	STRCLASS	3401	Two-Byte Signed Integer

Not used

53	RESERVED	3503	Two-Byte Signed Integer

This record type was used for NUMTYPES but was not required.

54	FORMAT	3602	Two-Byte Signed Integer

Defines the format of a Stream tape in two bytes. The possible values are:

1. for GDSII Archive format
2. for GDSII Filtered format
3. for EDSM Archive format
4. for EDSHI Filtered fornnat

An Archive Stream file contains elements for all the layers and data types. In an Archive Stream file, the FORMAT record is followed immediately by the UNITS record. A file which does not have the

FORMAT record is assumed to be an Archive file.

A Filtered Stream file contains only the elements on the layers and with the datatypes you specify during creation ofthe Stream file. The list of layers and datatypes specified appear in MASK records. At least one MASK record must immediately follow the FORMAT record. The MASK records are terminated with the ENDMASKS record.

55	MASK	3706	ASCII string

(Required for and present only in FilteredStream file.)
Contains the list of layers and datatypes specified by the user when creating the file. At least one MASK record must immediately follow the FORMAT record. More than one MASK record may occur. The last MASK record is followed by the ENDMASK record.
In the MASK list, datatypes are separated from the layers with a semicolon. Individual layers or datatypes are sepa-rated with a space. A range of layers or datatypes is specified with a dash.

An example MASK list looks like this: 1 5 -7 10 ; 0- 255

56	ENDMASKS	3800	NoData

(Required for and present only in FilteredStream file.)
Marks the end of the MASK records. The ENDMASKS record must follow the last MASK record. ENDMASKS is immediately followed by the UNITS record.

Welcome to the JPEG Tutorial!

This page presents a brief description of how JPEG compresses images. JPEG, unlike other formats like PPM, PGM, and GIF, is a ***lossy*** compression technique; this means visual information is lost permanently. The key to making JPEG work is choosing what data to throw away.

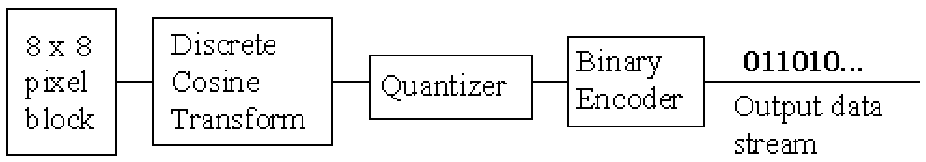
Introduction

JPEG is the image compression standard developed by the Joint Photographic Experts Group. It works best on natural images (scenes). This tutorial describes general JPEG compression for greyscale images; however, JPEG compresses color images just as easily. For instance, It compresses the red-green-blue parts of a color image as three separate greyscale images - each compressed to a different extent, if desired. The section, comparison of JPEG images, displays both color and black and white images compressed with JPEG.

How JPEG works

Figure one describes the JPEG process. JPEG divides up the image into 8 by 8 pixel blocks, and then calculates the discrete cosine transform (DCT) of each block. A quantizer rounds off the DCT coefficients according to the quantization matrix. This step produces the "lossy" nature of JPEG, but allows for large compression ratios. JPEG's compression technique uses a variable length code on these coefficients, and then writes the compressed data stream to an output file (*.jpg). For decompression, JPEG recovers the quantized DCT coefficients from the compressed data stream, takes the inverse transforms and displays the image. Figure 1 shows this process.

FIGURE 1. BLOCK DIAGRAM OF JPEG COMPRESSION



Testing Methods and Results

What does JPEG look like? How far can one compress an image and have it look identical to the original? presentable? merely recognizable? The comparison of JPEG images answers these questions. This section tests color and b/w pictures compressed at different levels. The original images are in PBM format. Version 3.0 of *XV, the imaging system for X-Windows,* implemented the JPEG compression.

References

Many books and articles discuss JPEG in more detail. An anonymous FTP site for more JPEG documentation is: ftp.uu.net/graphics/jpeg/.

Index of Hyperlinks

Below are all the hyperlinks used in this document.

Comments or Problems

Thank you for reading this JPEG tutorial. If you have any problems with this document, or would like more information, please send email to ace@ecn.purdue.edu.
