INTERNATIONAL TELECOMMUNICATION UNION

CCITT

T.81

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

(09/92)

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.

© ITU 1993

All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the ITU.

Contents

Page

Introduction iii

1 Scope 1

2 Normative references 1

3 Definitions, abbreviations and symbols 1

4 General 12

5 Interchange format requirements 23

6 Encoder requirements 23

7 Decoder requirements 23

Annex A – Mathematical definitions 24

Annex B – Compressed data formats 31

Annex C – Huffman table specification 50

Annex D – Arithmetic coding 54

Annex E – Encoder and decoder control procedures 77

Annex F – Sequential DCT-based mode of operation 87

Annex G – Progressive DCT-based mode of operation 119

Annex H – Lossless mode of operation 132

Annex J – Hierarchical mode of operation 137

Annex K – Examples and guidelines 143

Annex L – Patents 179

Annex M – Bibliography 181

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.

Symbol	Meaning
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*.

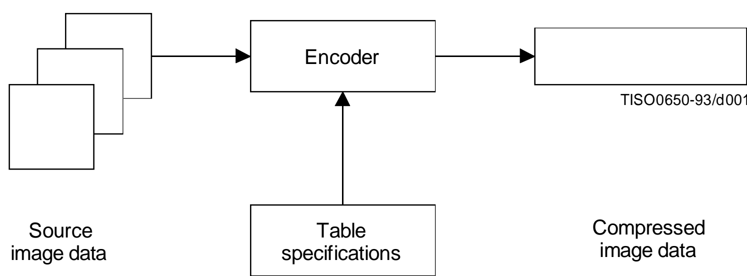


Figure 1 – Encoder

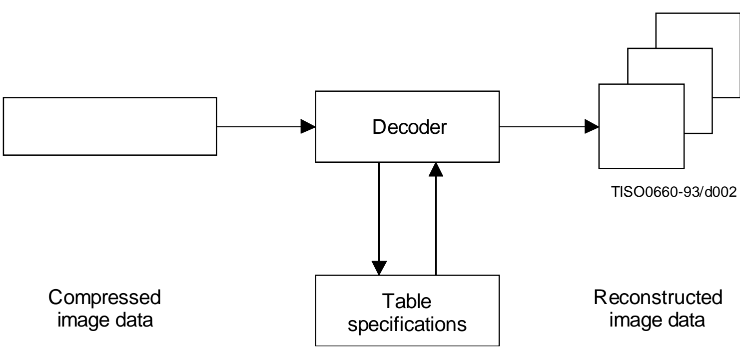


Figure 2 – Decoder

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.

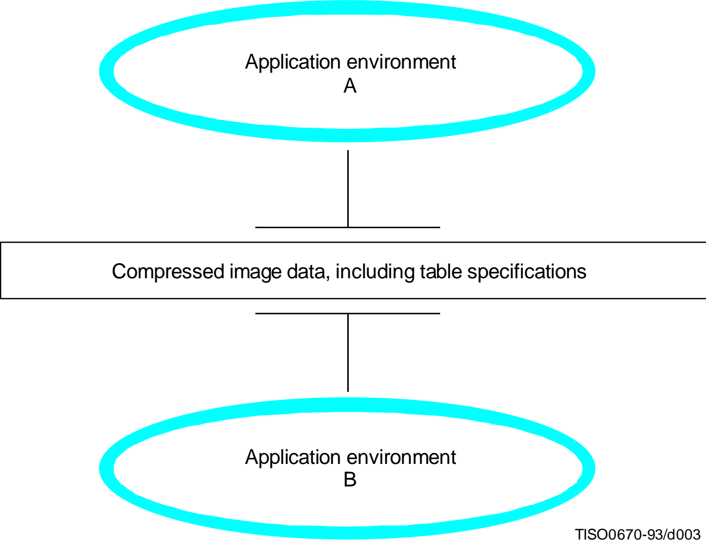


Figure 3 – Interchange format for compressed image data

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.

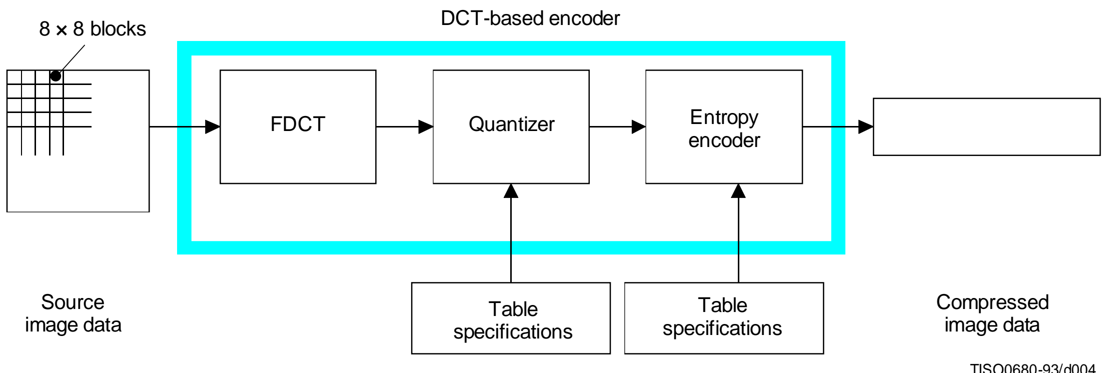


Figure 4 – DCT-based encoder simplified diagram

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.

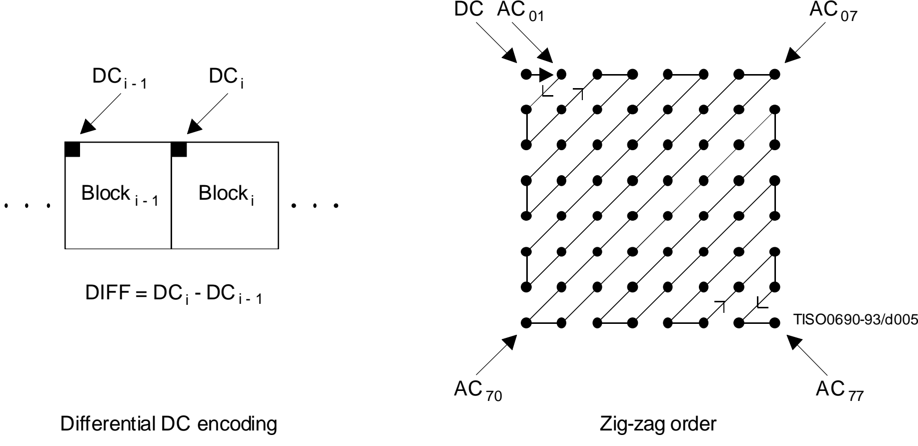


Figure 5 – Preparation of quantized coefficients for entropy encoding

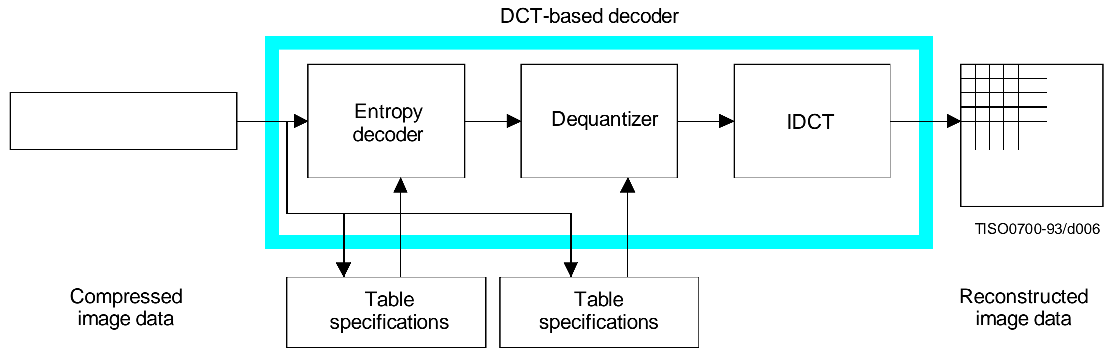


Figure 6 – DCT-based decoder simplified diagram

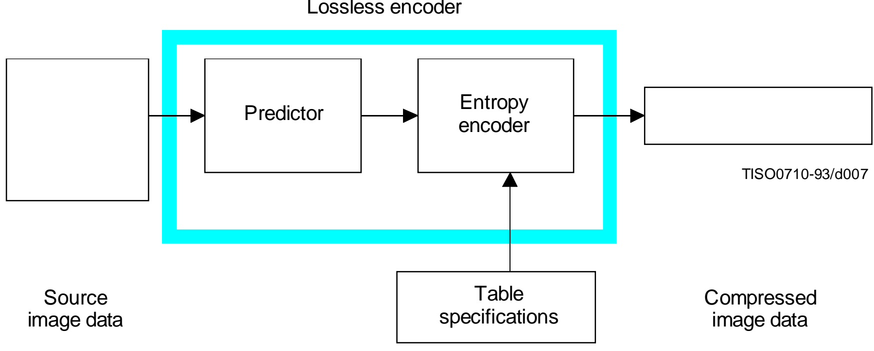


Figure 7 – Lossless encoder simplified diagram

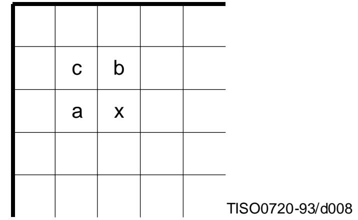


Figure 8 – 3-sample prediction neighbourhood

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.

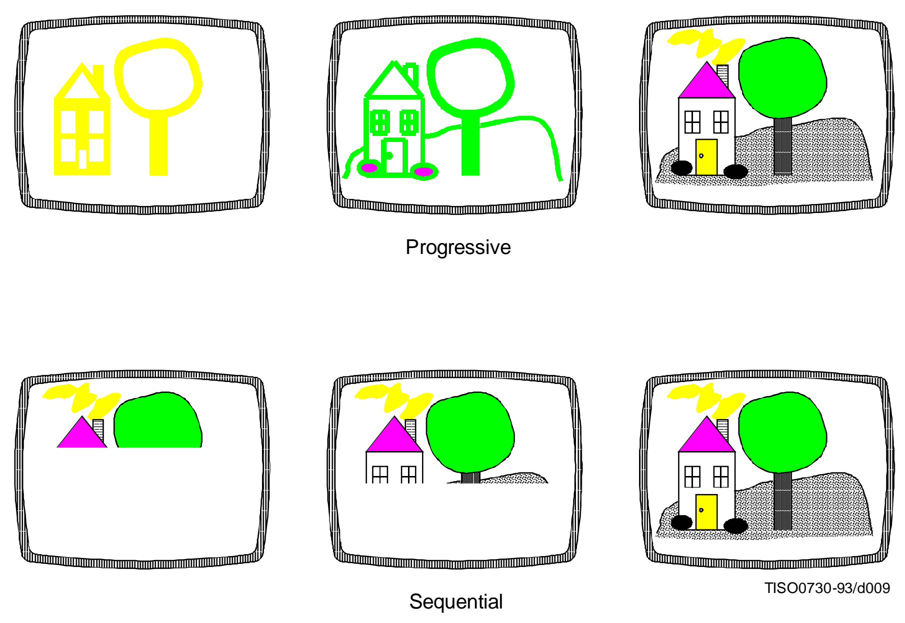


Figure 9 – Progressive versus sequential presentation

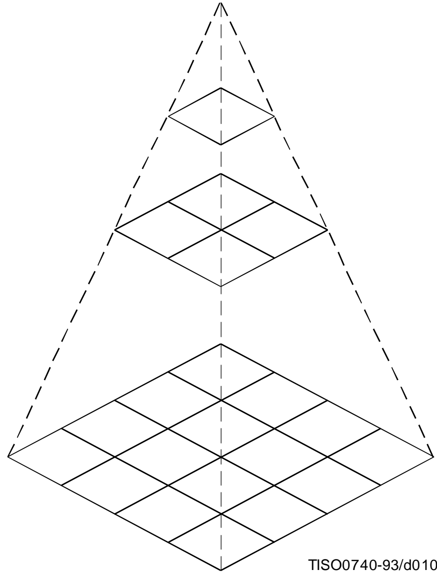


Figure 10 – Hierarchical multi-resolution encoding

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.)

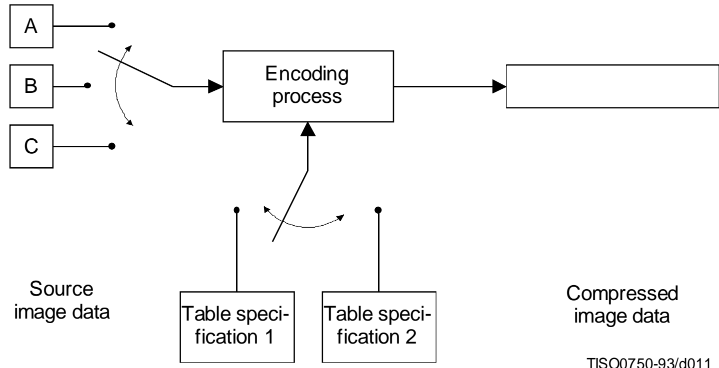


Figure 11 – Component-interleave and table-switching control

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.

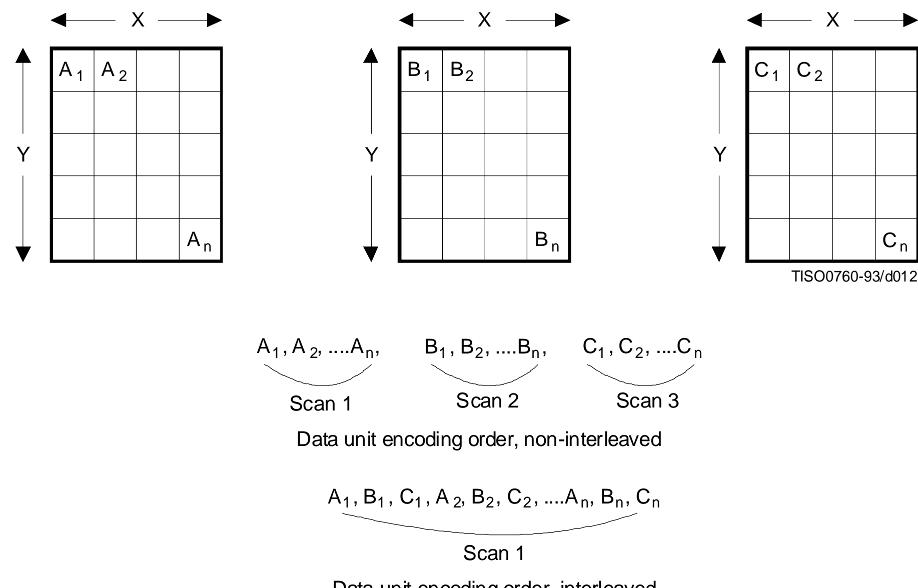


Figure 12 – Interleaved versus non-interleaved encoding order

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.)

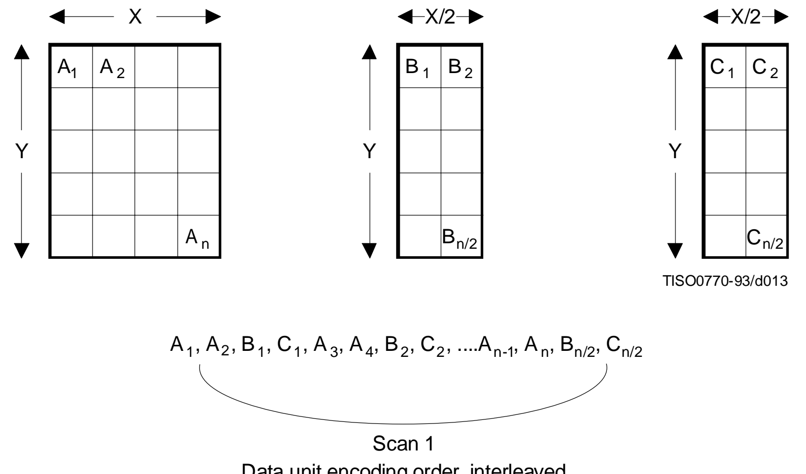


Figure 13 – Interleaved order for components with different dimensions

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

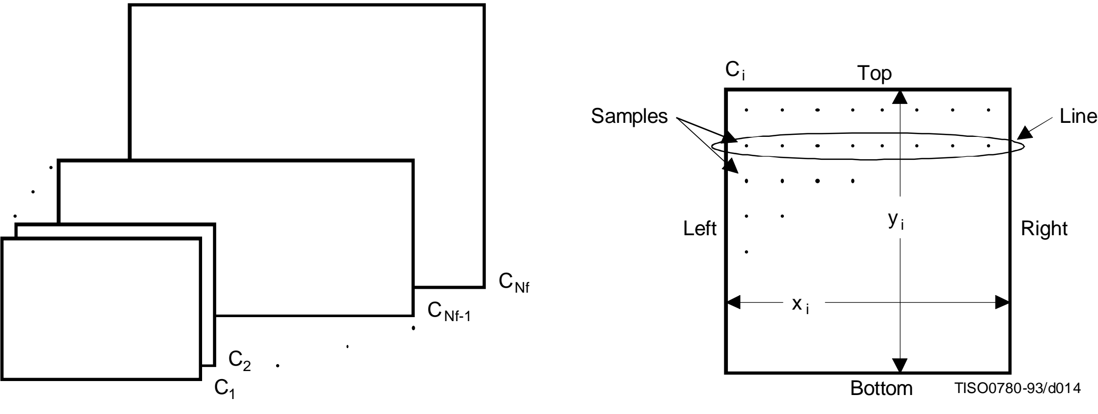
A sample is an integer with precision P bits, with any value in the range 0 through 2^{P-1}. All samples of all components within an image shall have the same precision P. Restrictions on the value of P depend on the mode of operation, as specified in B.2 to B.7.

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.

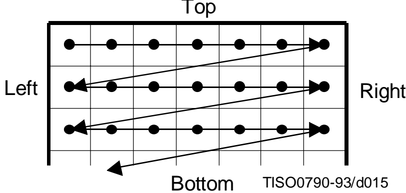


Figure A.2 – Non-interleaved data ordering

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.

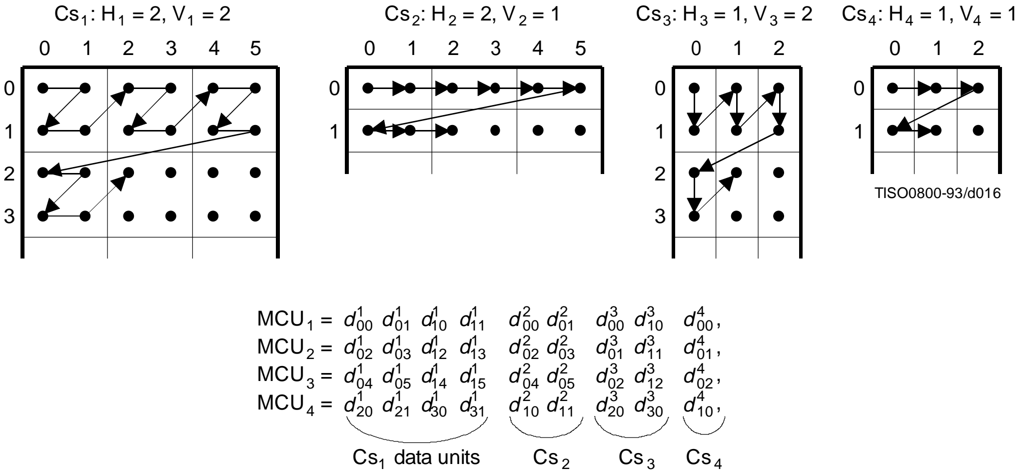


Figure A.3 – Interleaved data ordering example

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.

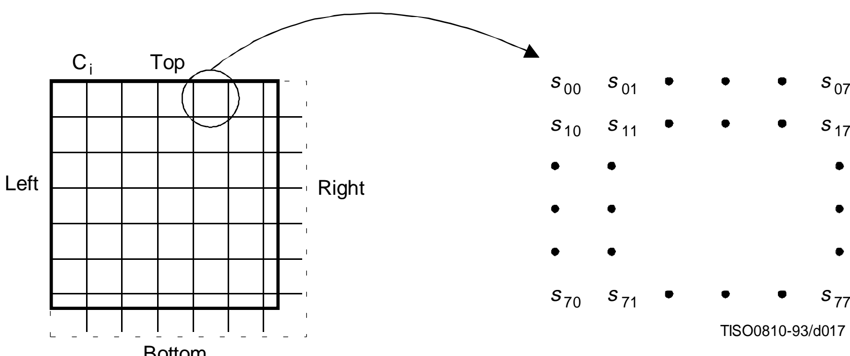


Figure A.4 – Partition and orientation of 8 x 8 sample blocks

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.

FDCT:
$$S_{vu} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{yx} \cos\frac{(2x+1)u\pi}{16} \cos\frac{(2y+1)v\pi}{16}$$

IDCT:
$$s_{yx} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C_u C_v S_{vu} \cos\frac{(2x+1)u\pi}{16} \cos\frac{(2y+1)v\pi}{16}$$

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.

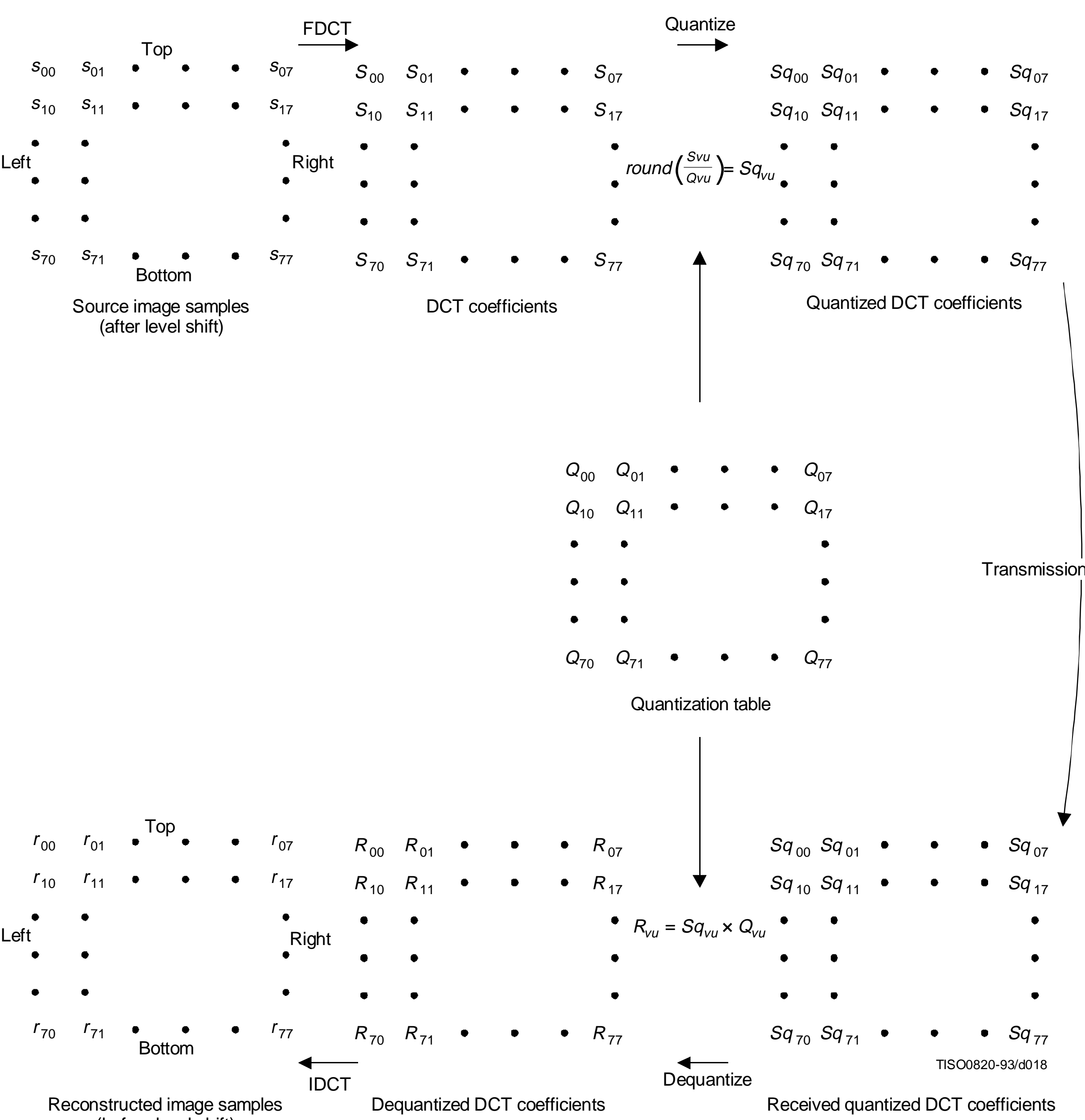


Figure A.5 – Relationship between 8 × 8-block samples and DCT coefficients

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' X'FFC1' X'FFC2' X'FFC3'	SOF_0 SOF_1 SOF_2 SOF_3	Baseline DCT Extended sequential DCT Progressive DCT Lossless (sequential)
Start Of Frame markers, differential, Huffman coding		
X'FFC5' X'FFC6' X'FFC7'	SOF_5 SOF_6 SOF_7	Differential sequential DCT Differential progressive DCT Differential lossless (sequential)
Start Of Frame markers, non-differential, arithmetic coding		
X'FFC8' X'FFC9' X'FFCA' X'FFCB'	JPG SOF_9 SOF_{10} SOF_{11}	Reserved for JPEG extensions Extended sequential DCT Progressive DCT Lossless (sequential)
Start Of Frame markers, differential, arithmetic coding		
X'FFCD' X'FFCE' X'FFCF'	SOF_{13} SOF_{14} SOF_{15}	Differential sequential DCT Differential progressive DCT 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' X'FFD9' X'FFDA' X'FFDB' X'FFDC' X'FFDD' X'FFDE' X'FFDF' X'FFE0' through X'FFEF' X'FFF0' through X'FFFD' X'FFFE'	SOI* EOI* SOS DQT DNL DRI DHP EXP APP_n JPG_n COM	Start of image End of image Start of scan Define quantization table(s) Define number of lines Define restart interval Define hierarchical progression Expand reference component(s) Reserved for application segments Reserved for JPEG extensions Comment
Reserved markers		
X'FF01' X'FF02' through X'FFBF'	TEM* RES	For temporary private use in arithmetic coding 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.

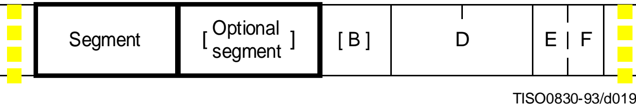


Figure B.1 – Syntax notation conventions

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.

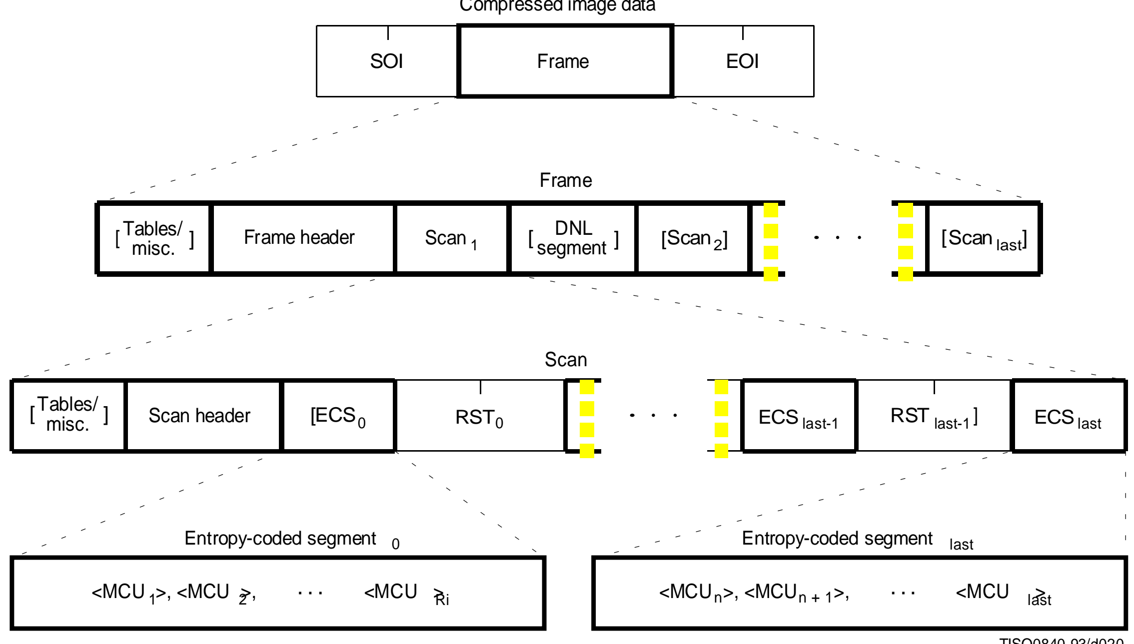


Figure B.2 – Syntax for sequential DCT-based, progressive DCT-based, and lossless modes of operation

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.

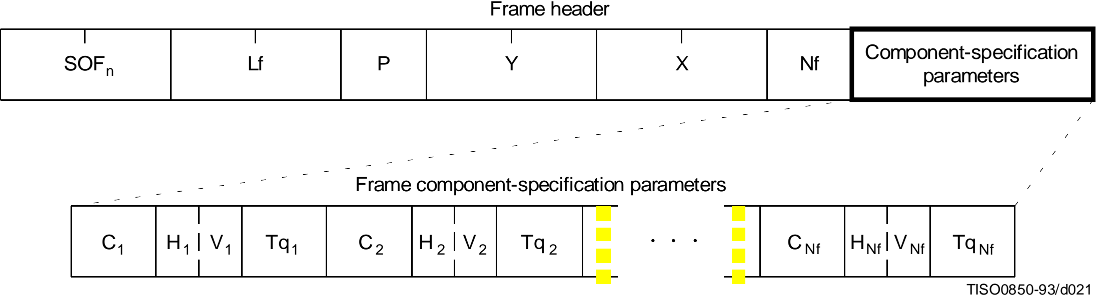


Figure B.3 – Frame header syntax

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 × 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.

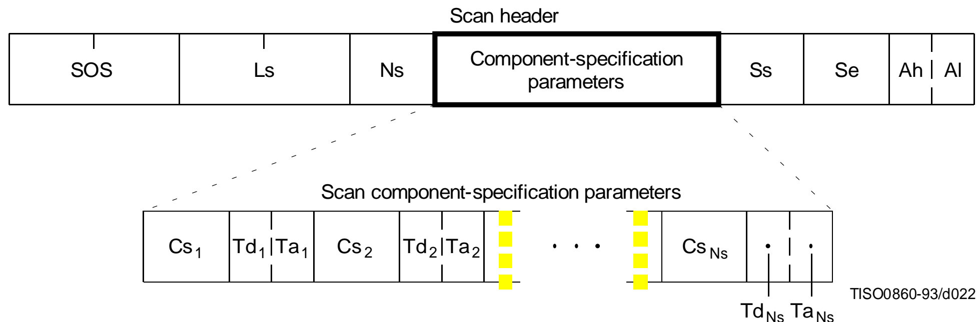


Figure B.4 – Scan header syntax

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.

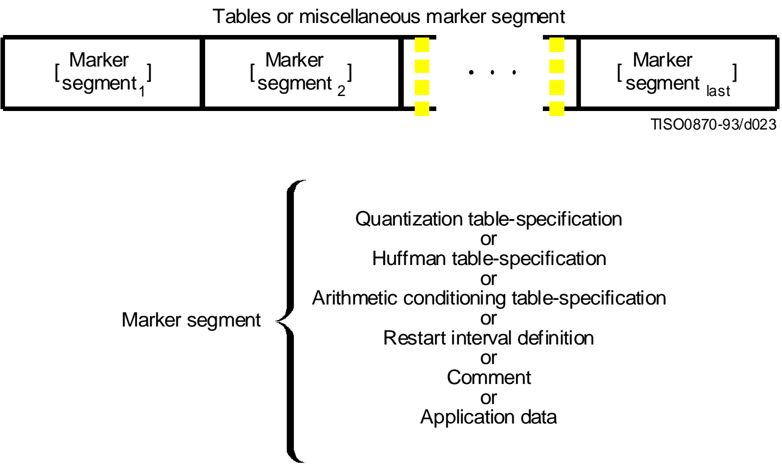


Figure B.5 – Tables/miscellaneous marker segment syntax

B.2.4.1 Quantization table-specification syntax

Figure B.6 specifies the marker segment which defines one or more quantization tables.

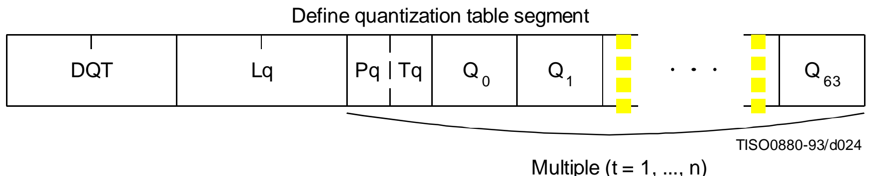


Figure B.6 – Quantization table syntax

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 kth 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.

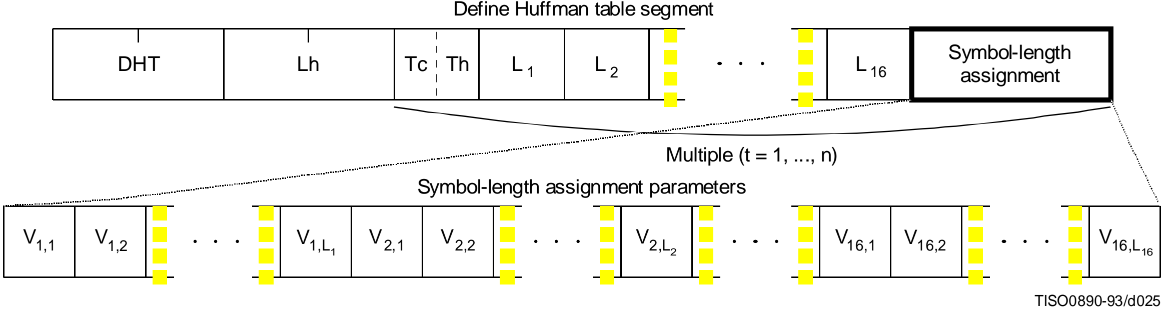


Figure B.7 – Huffman table syntax

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.)

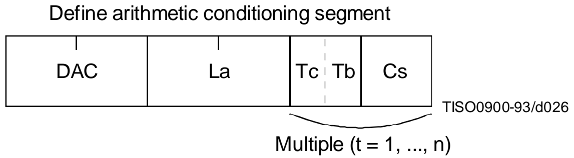


Figure B.8 – Arithmetic conditioning table-specification syntax

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 $1 \le Kx \le 63$. 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 $0 \le L \le U \le 15$ 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	$2 + 2 \times n$		
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.

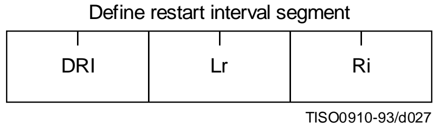


Figure B.9 – Restart interval definition syntax

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 × MCUR

B.2.4.5 Comment syntax

Figure B.10 specifies the marker segment structure for a comment segment.

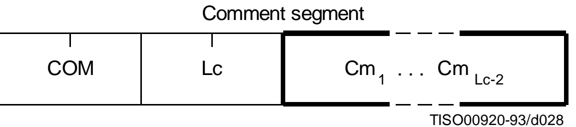


Figure B.10 – Comment segment syntax

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.

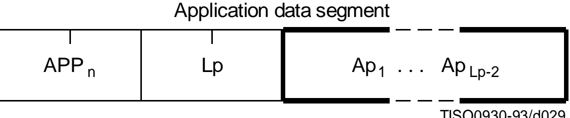


Figure B.11 – Application data syntax

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.

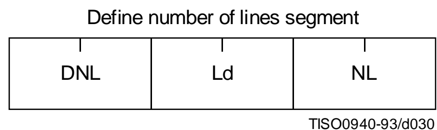


Figure B.12 – Define number of lines syntax

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.

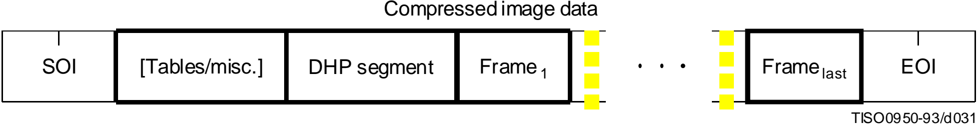


Figure B.13 – Syntax for the hierarchical mode of operation

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.

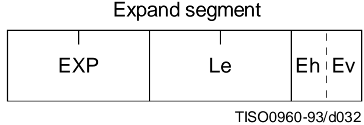


Figure B.14 – Syntax of the expand segment

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).

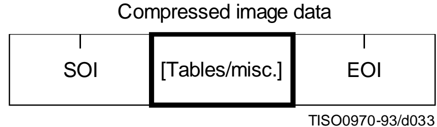


Figure B.15 – Abbreviated format for table-specification data syntax

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.

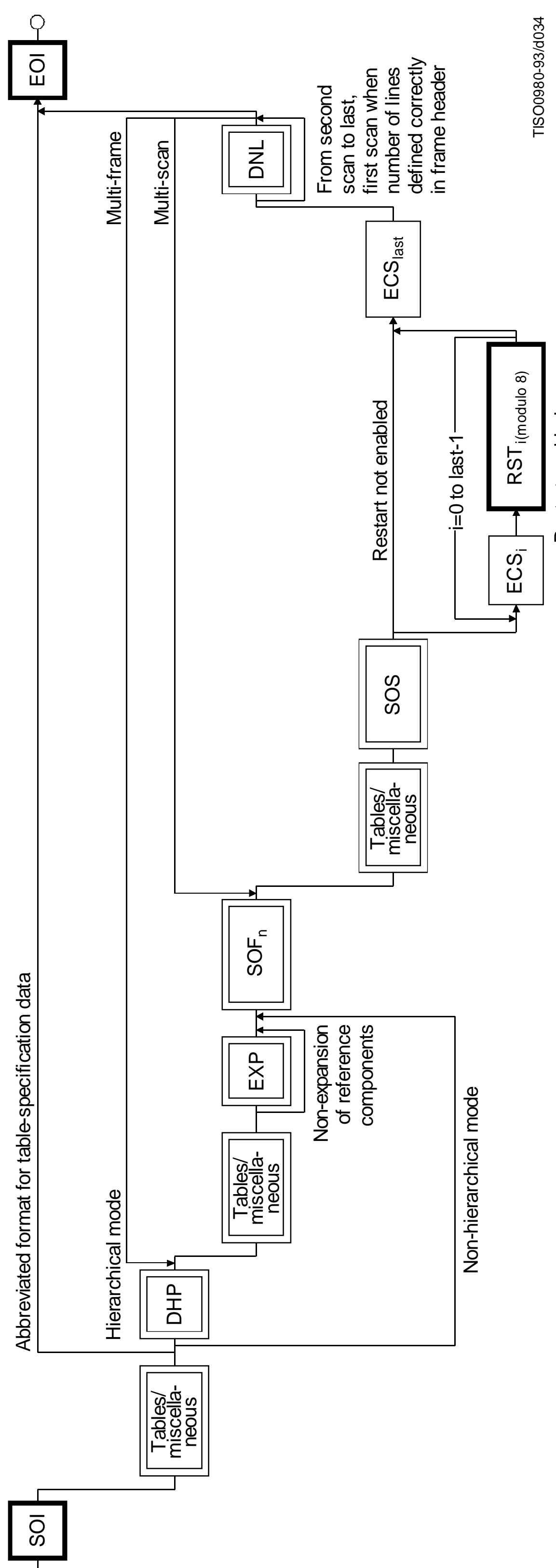


Figure B.16 – Flow of compressed data syntax

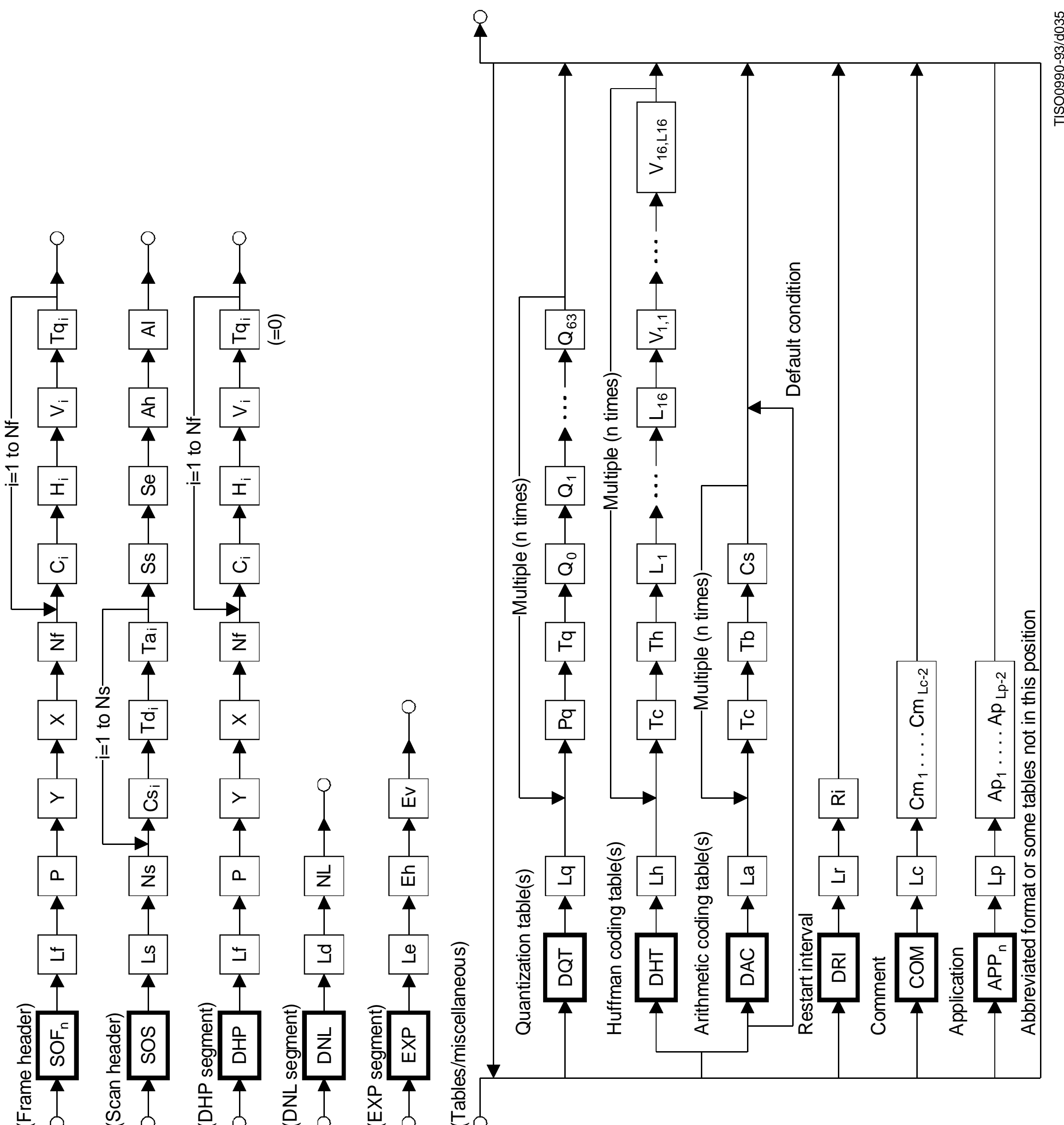


Figure B.17 – Flow of marker segment

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.

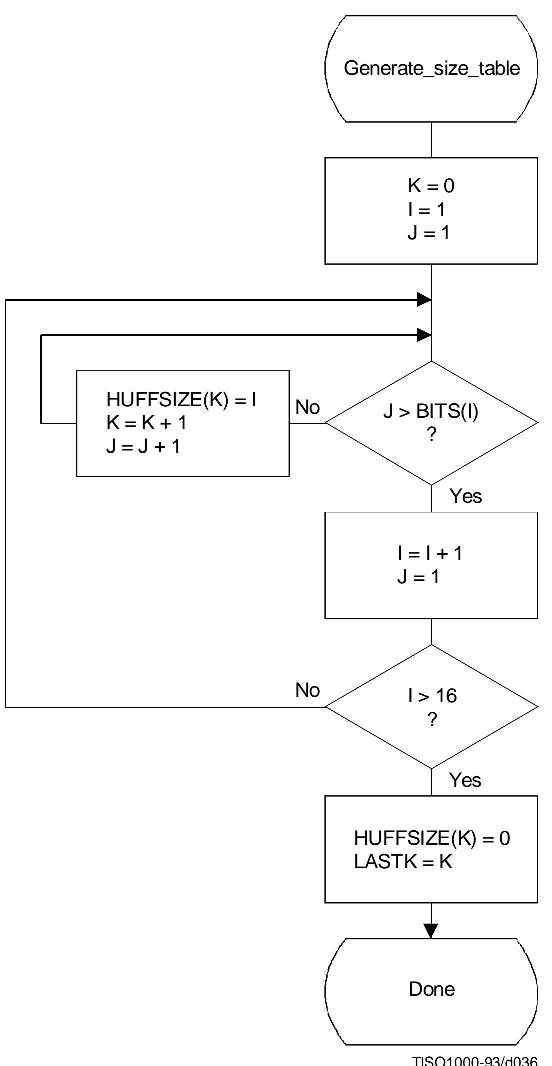


Figure C.1 – Generation of table of Huffman code sizes

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.

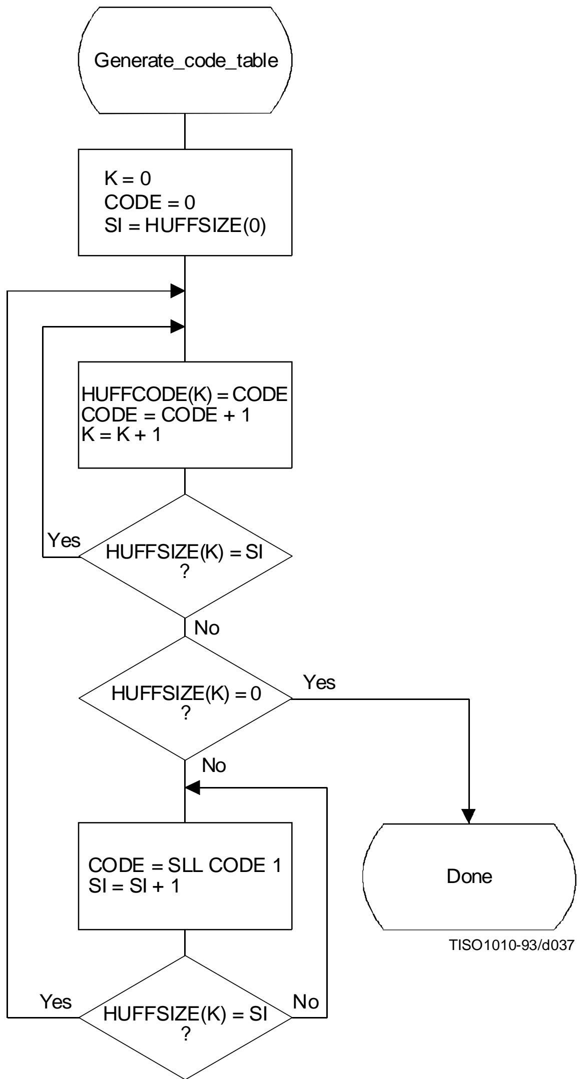


Figure C.2 – Generation of table of Huffman codes

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.

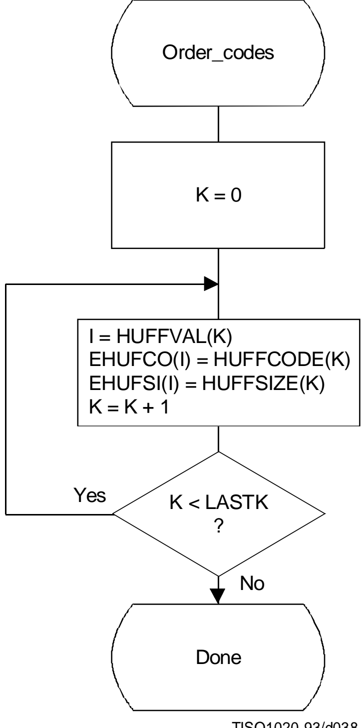


Figure C.3 – Ordering procedure for encoding procedure code tables

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' ≤ 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 ≤ 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 ≤ 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) × A	Probability sub-interval for the LPS;
A – (Qe(S) × 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.

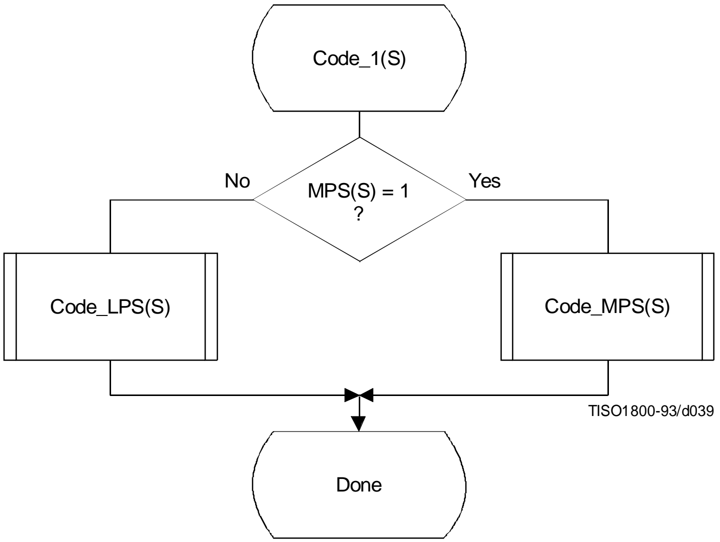


Figure D.1 – Code_1(S) procedure

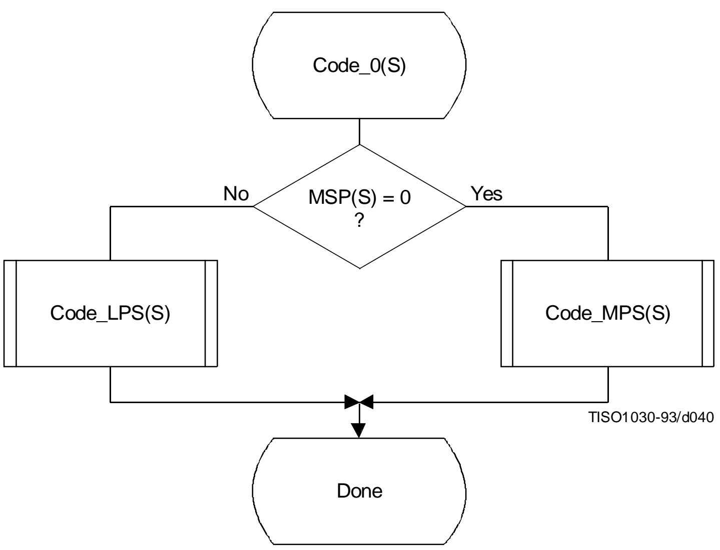


Figure D.2 – Code_0(S) procedure

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).

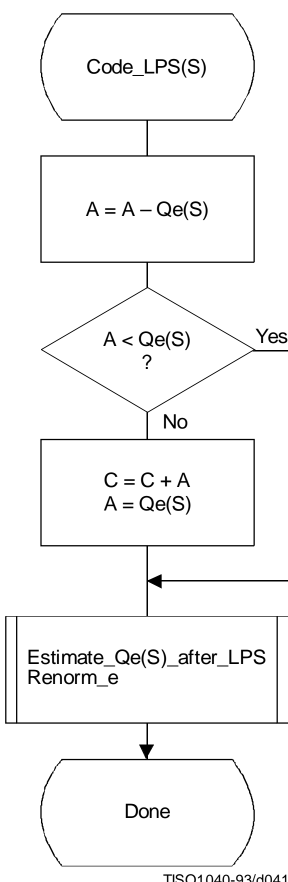


Figure D.3 – Code_LPS(S) procedure with conditional MPS/LPS exchange

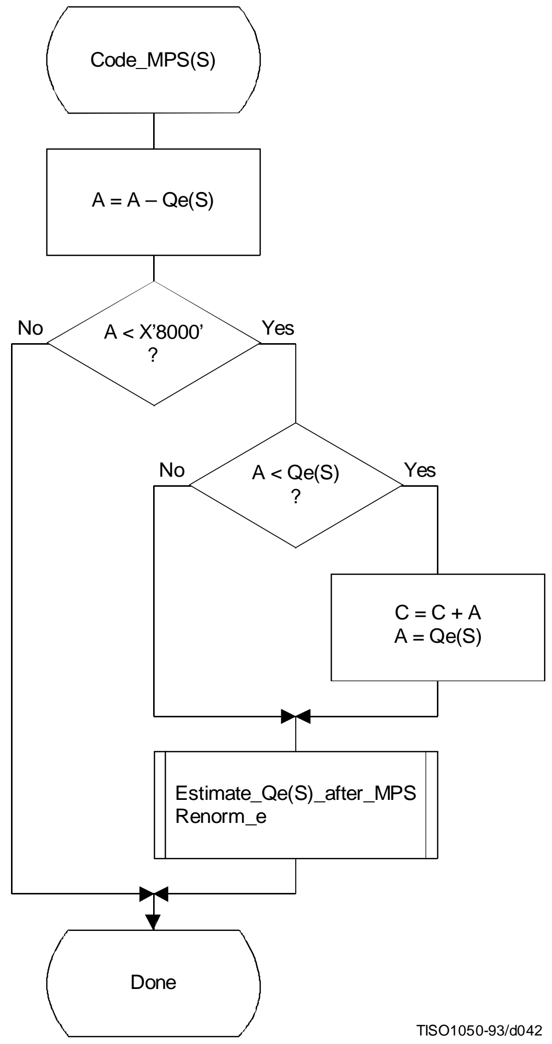


Figure D.4 – Code_MPS(S) procedure with conditional MPS/LPS exchange

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 $(4/3) \times (X'8000')$.

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.

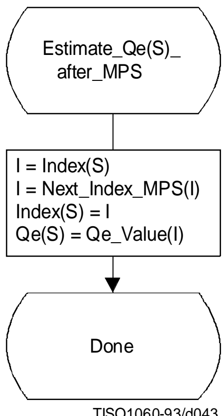


Figure D.5 – Probability estimation on MPS renormalization path

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.

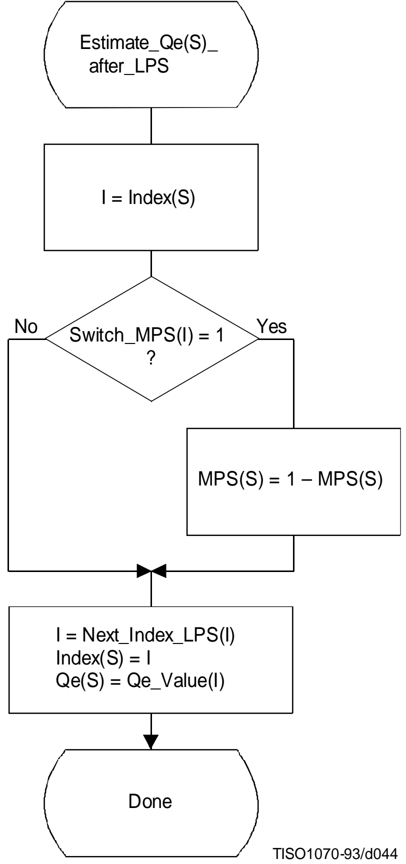


Figure D.6 – Probability estimation on LPS renormalization path

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'.

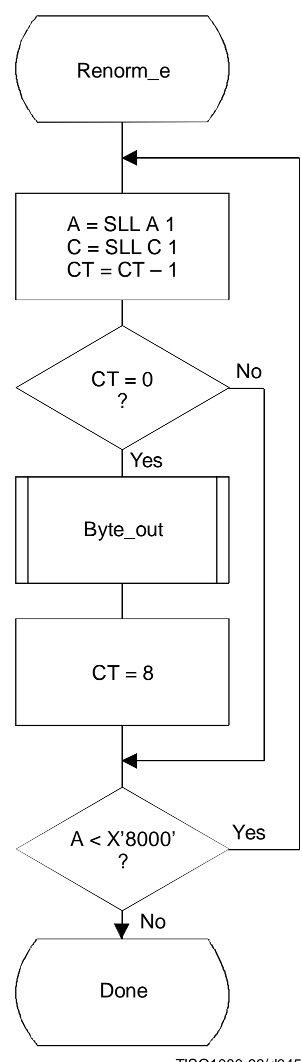


Figure D.7 – Encoder renormalization procedure

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.

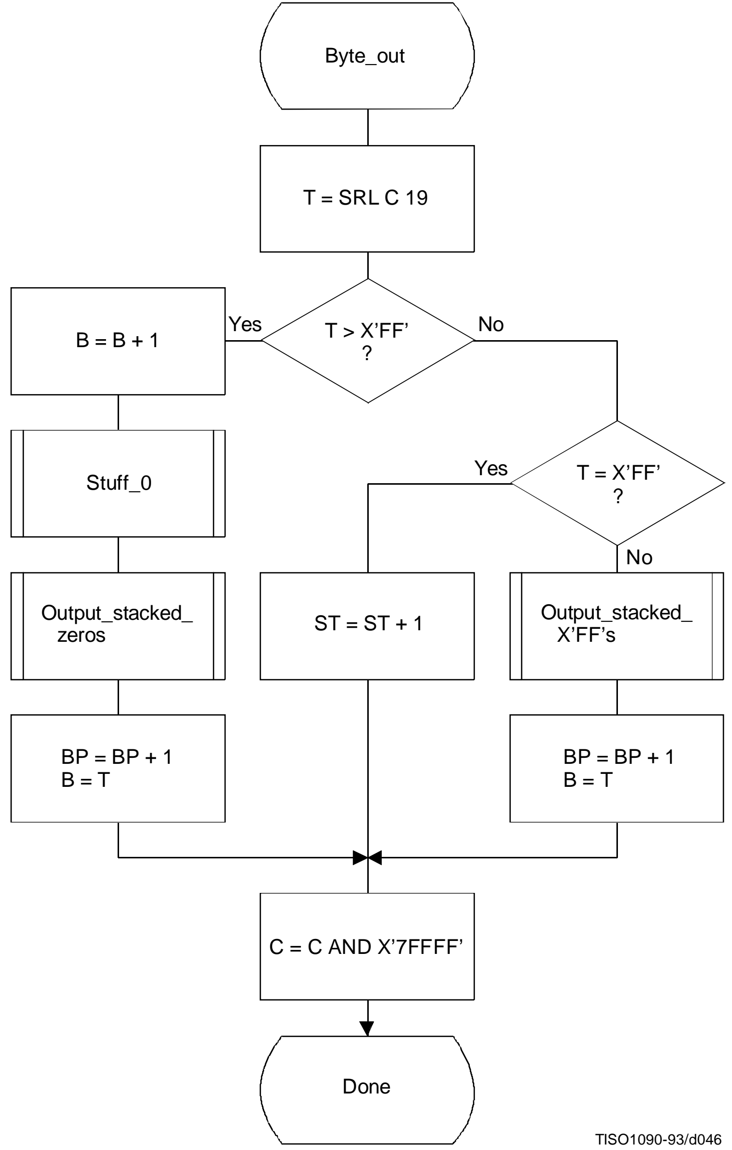


Figure D.8 – Byte_out procedure for encoder

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.

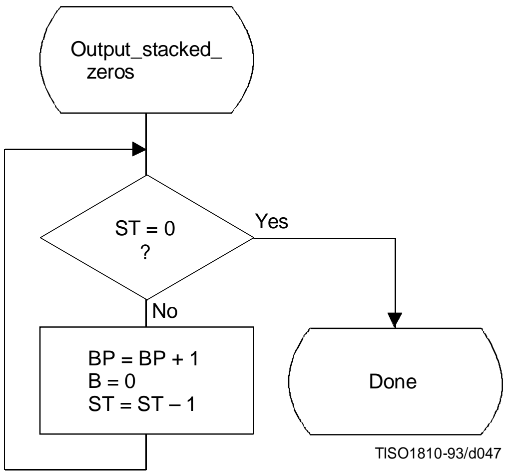


Figure D.9 – Output_stacked_zeros procedure for encoder

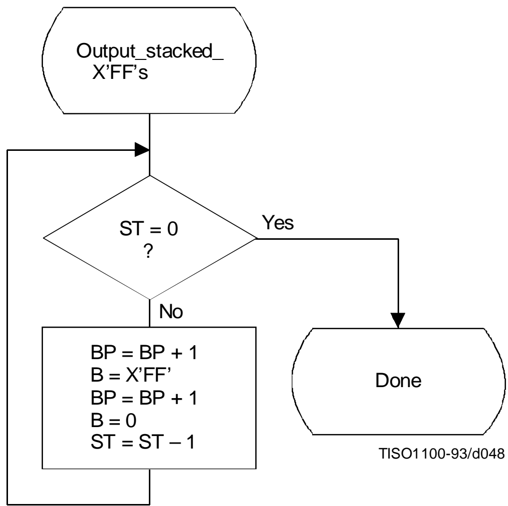


Figure D.10 – Output_stacked_X'FF's procedure for encoder

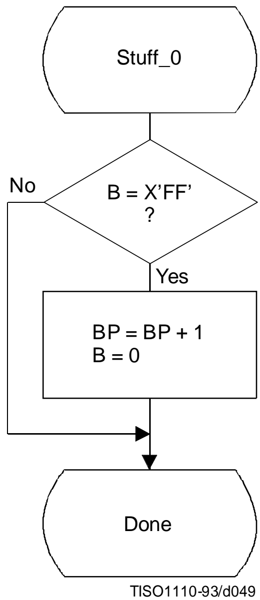


Figure D.11 – Stuff_0 procedure for encoder

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.

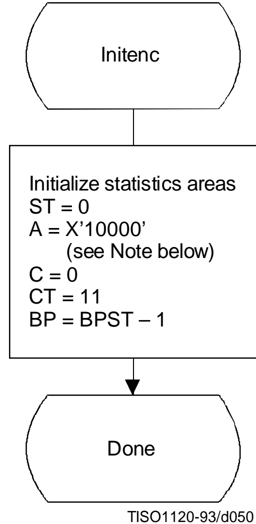


Figure D.12 – Initialization of the encoder

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.

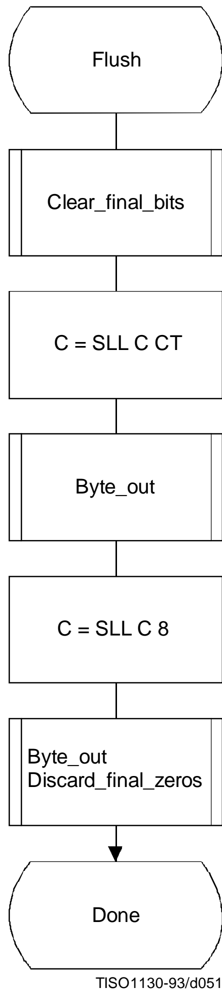


Figure D.13 – Flush procedure

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.

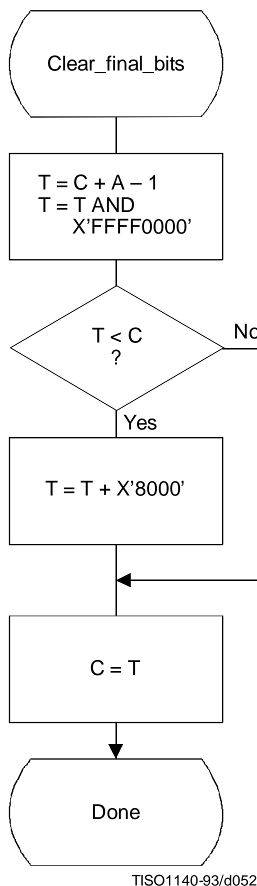


Figure D.14 – Clear_final_bits procedure in Flush

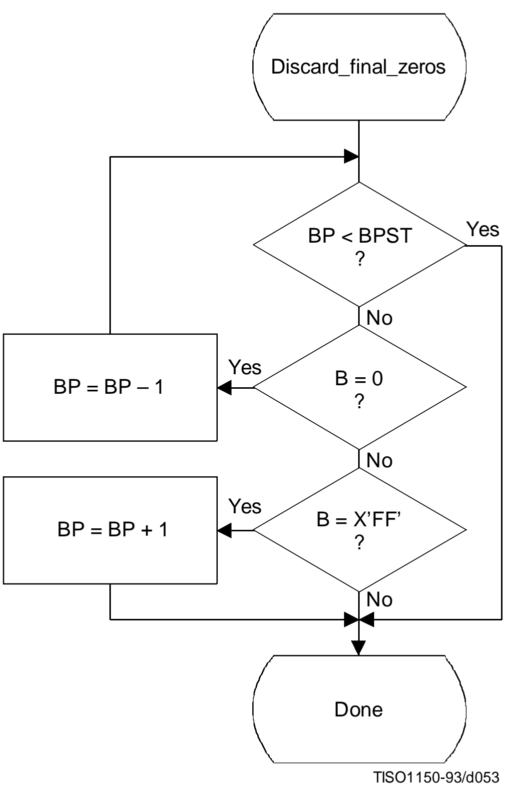


Figure D.15 – Discard_final_zeros procedure in Flush

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).

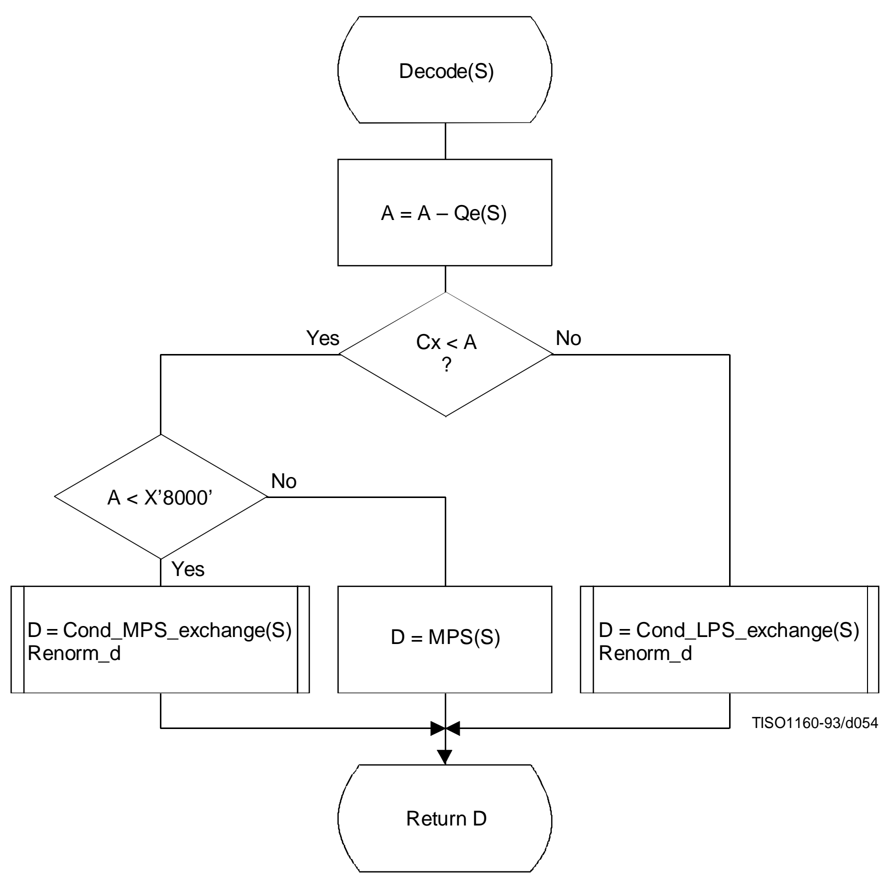


Figure D.16 – Decode(S) procedure

For the MPS path of the decoder the conditional exchange procedure is given in Figure D.18.

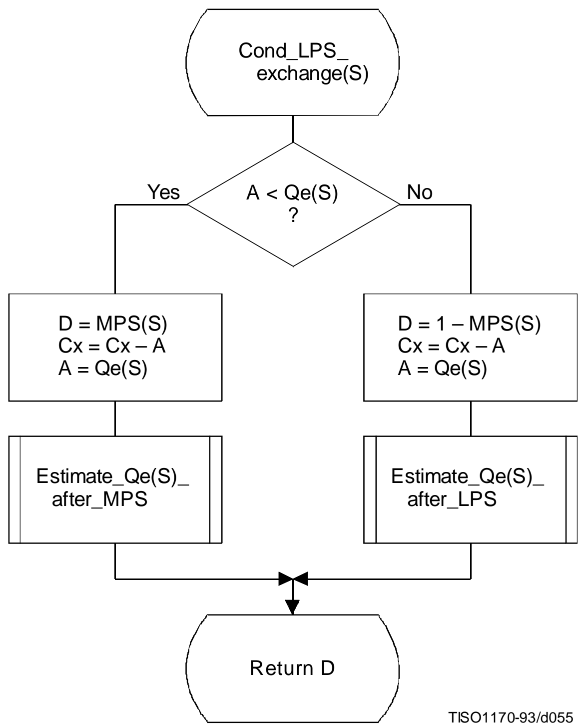


Figure D.17 – Decoder LPS path conditional exchange procedure

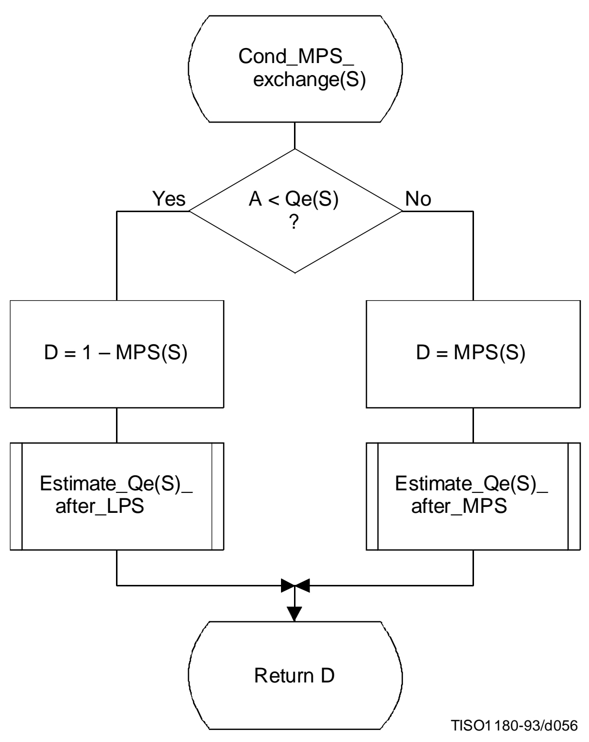


Figure D.18 – Decoder MPS path conditional exchange procedure

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'.

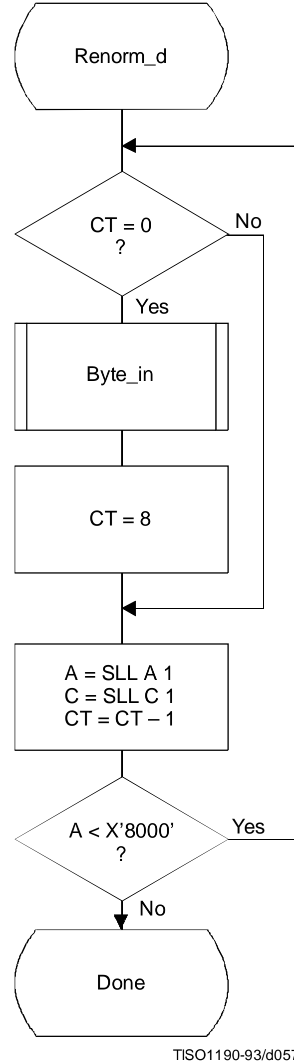


Figure D.19 – Decoder renormalization procedure

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.

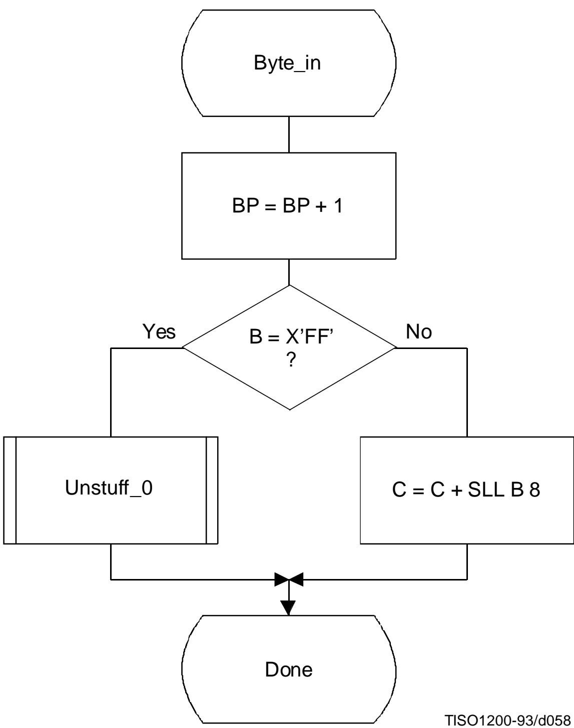


Figure D.20 – Byte_in procedure for decoder

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.

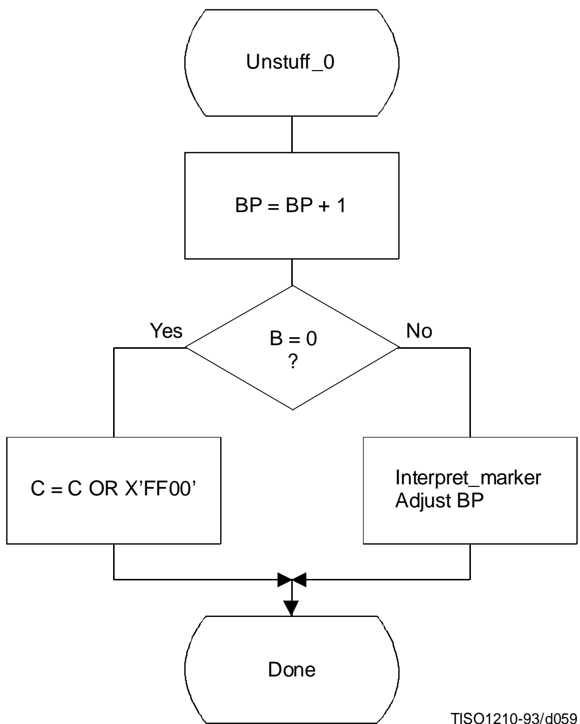


Figure D.21 – Unstuff_0 procedure for decoder

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.

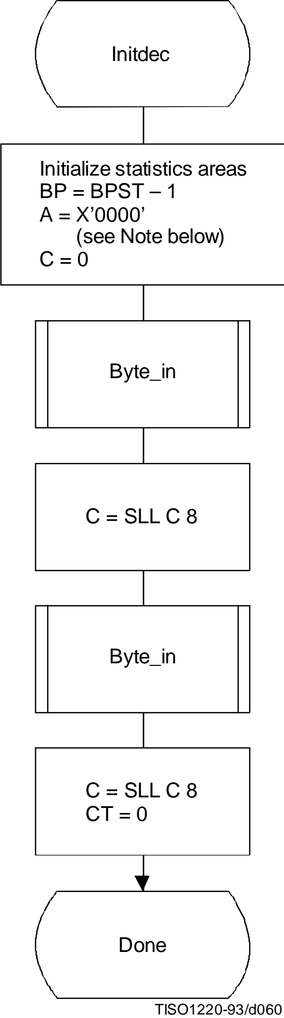


Figure D.22 – Initialization of the decoder

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.

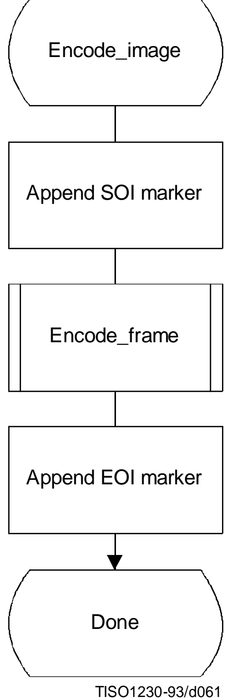


Figure E.1 – Control procedure for encoding an image

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.

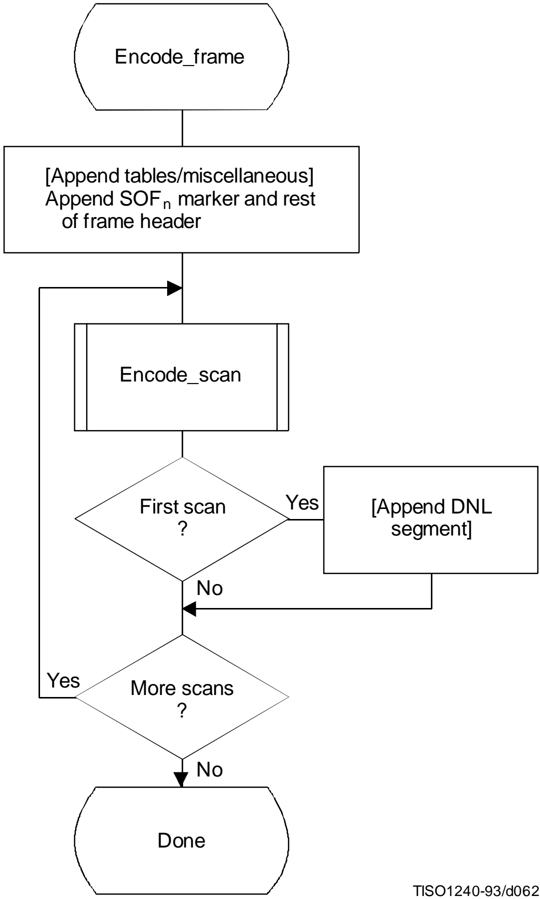


Figure E.2 – Control procedure for encoding a frame

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.

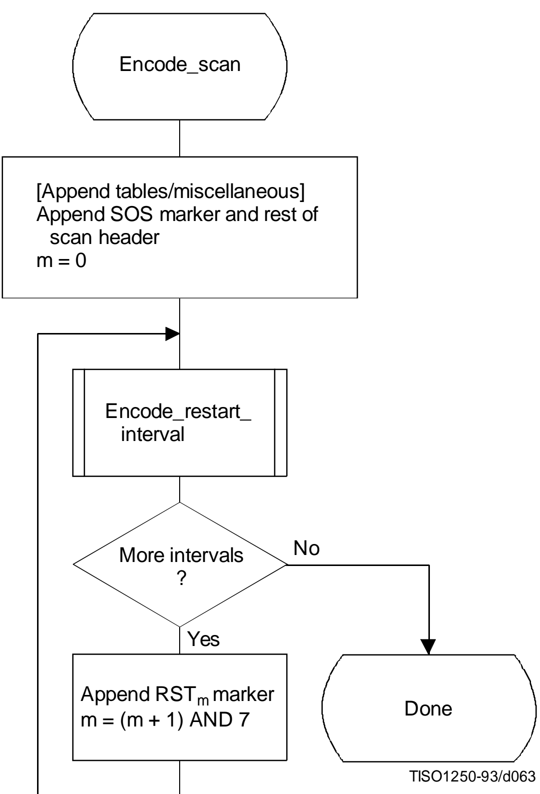


Figure E.3 – Control procedure for encoding a scan

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.

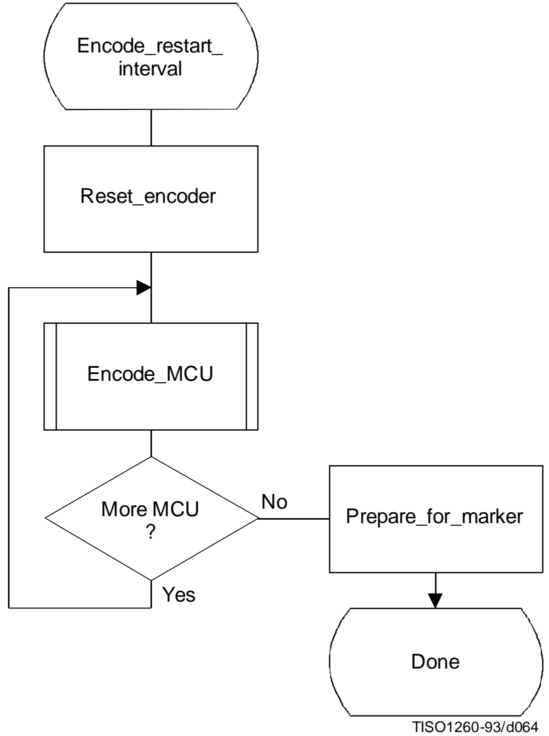


Figure E.4 – Control procedure for encoding a restart interval

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.

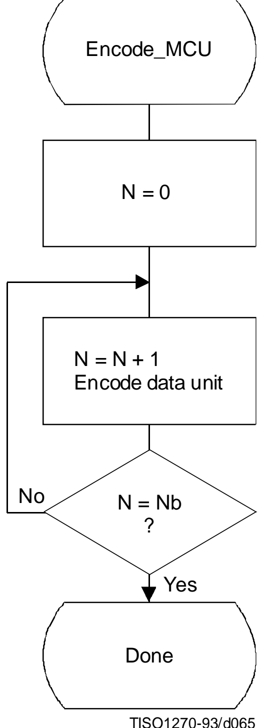


Figure E.5 – Control procedure for encoding a minimum coded unit (MCU)

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 SOF_n (Start Of Frame) marker; if this is not found, one of the marker segments listed in Table E.1 has been received.

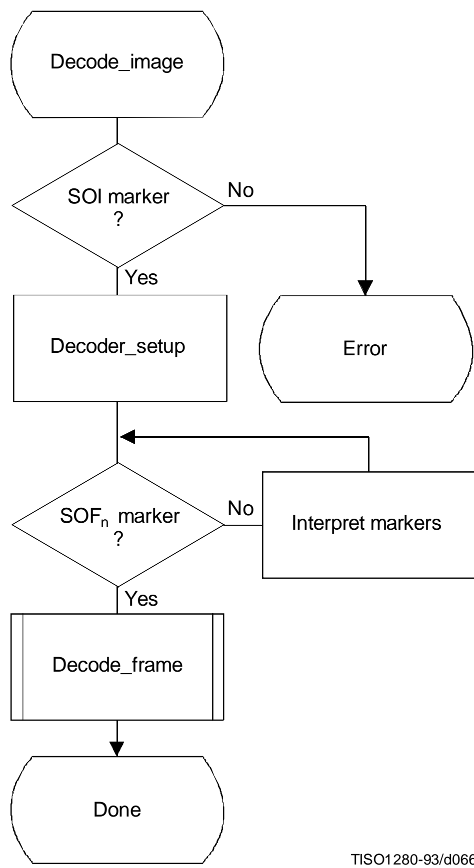


Figure E.6 – Control procedure for decoding compressed image data

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.

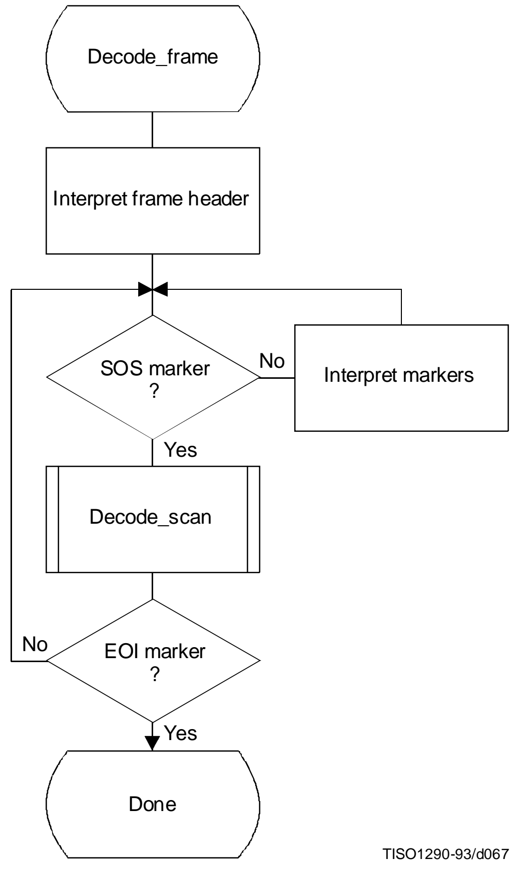


Figure E.7 – Control procedure for decoding a frame

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.

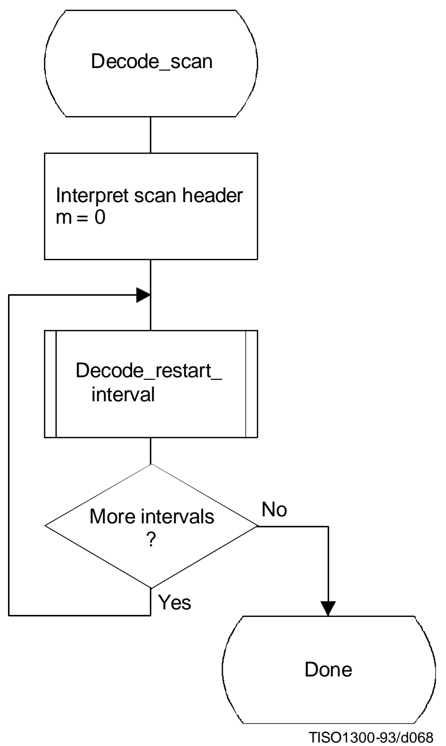


Figure E.8 – Control procedure for decoding a scan

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.

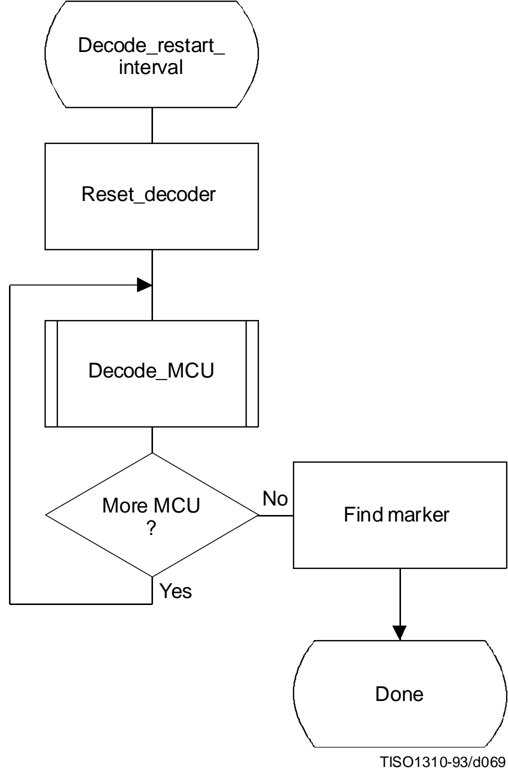


Figure E.9 – Control procedure for decoding a restart interval

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.

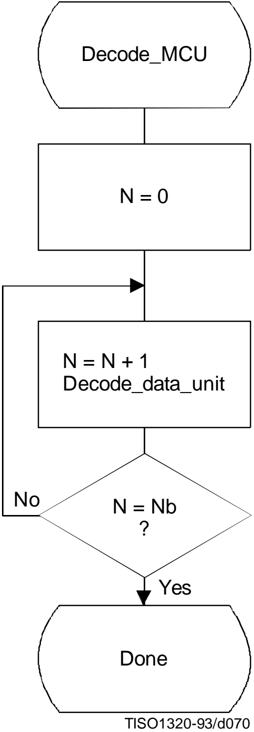


Figure E.10 – Control procedure for decoding a minimum coded unit (MCU)

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.

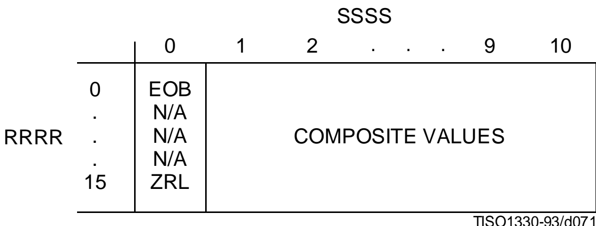


Figure F.1 – Two-dimensional value array for Huffman coding

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.

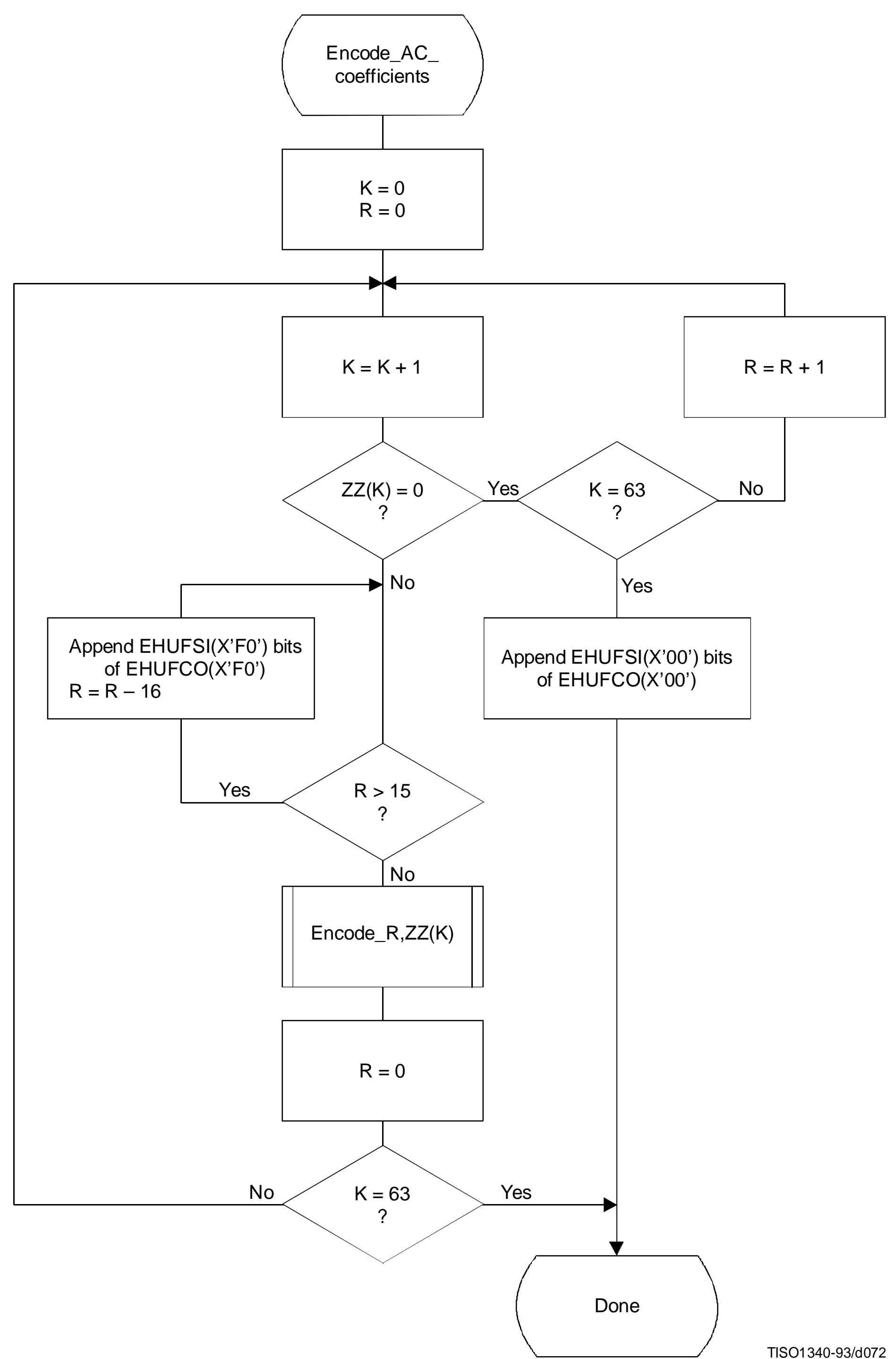


Figure F.2 – Procedure for sequential encoding of AC coefficients with Huffman coding

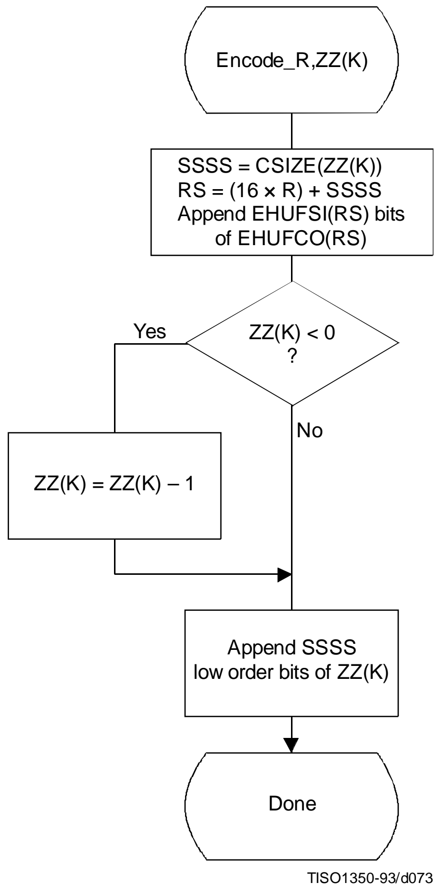


Figure F.3 – Sequential encoding of a non-zero AC coefficient

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.

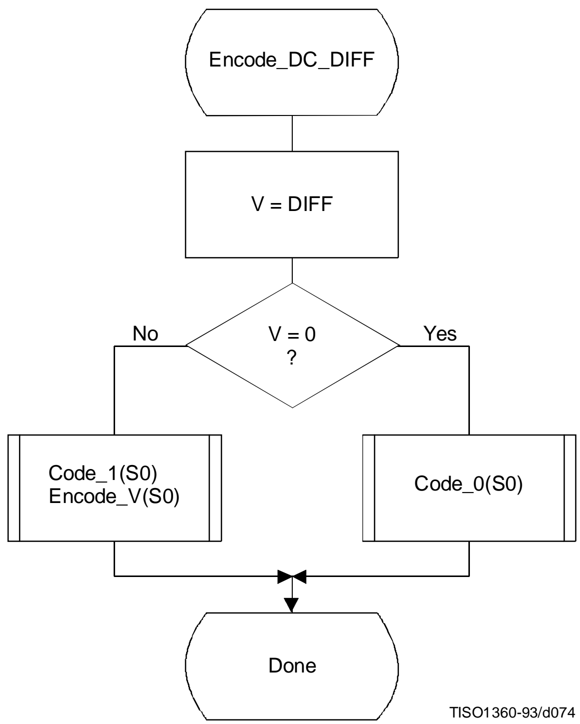


Figure F.4 – Coding model for arithmetic coding of DC difference

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.

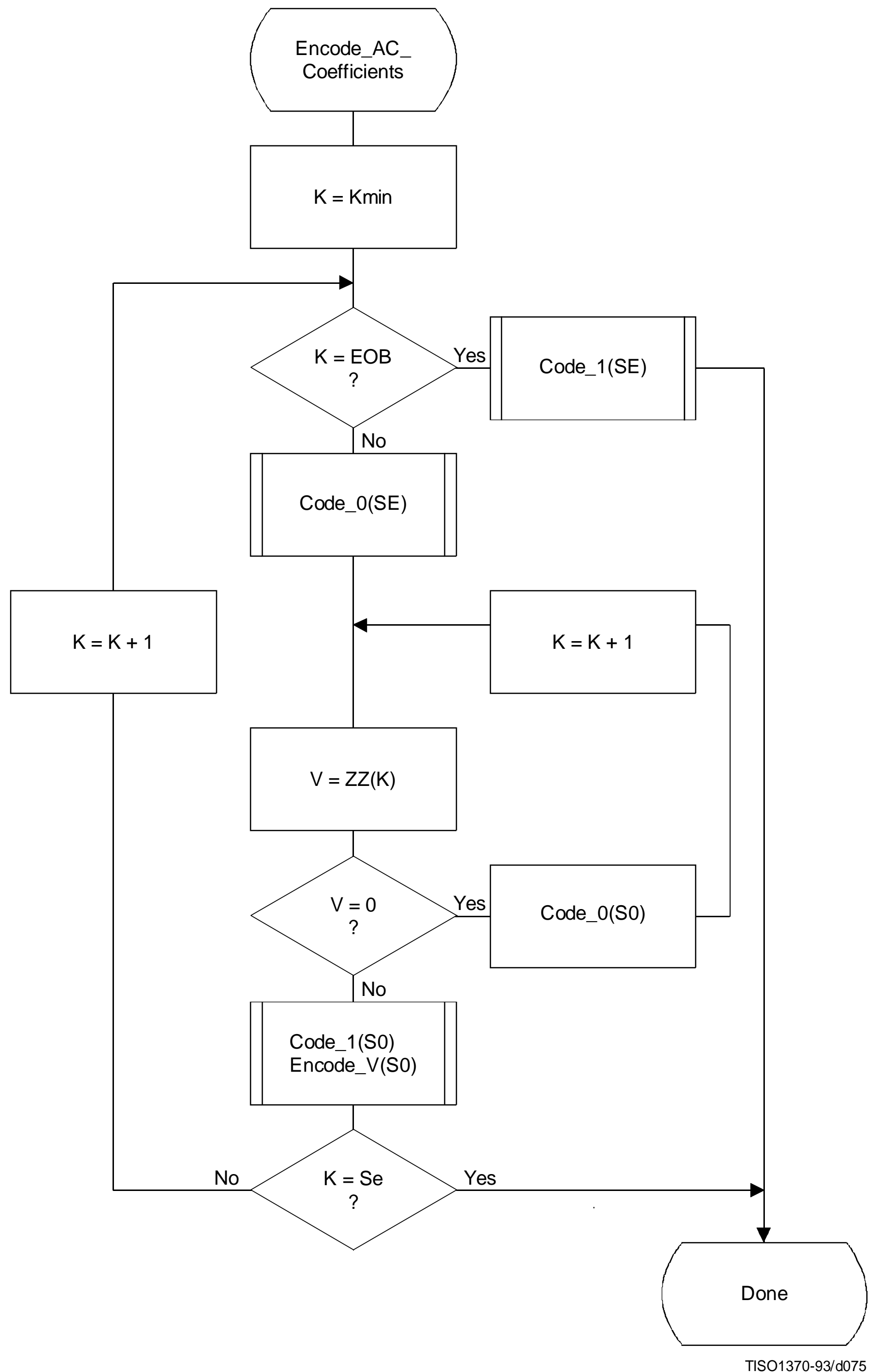


Figure F.5 – AC coding model for arithmetic coding

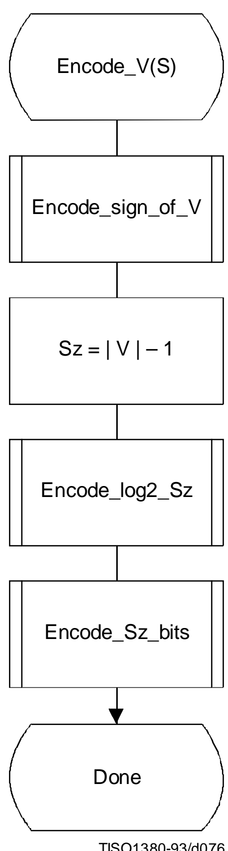


Figure F.6 – Sequence of procedures in encoding non-zero values of V

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.

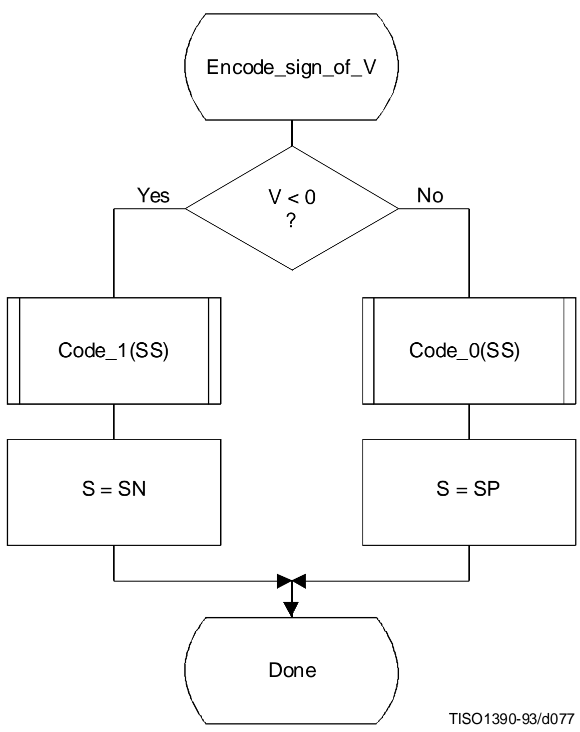


Figure F.7 – Encoding the sign of V

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

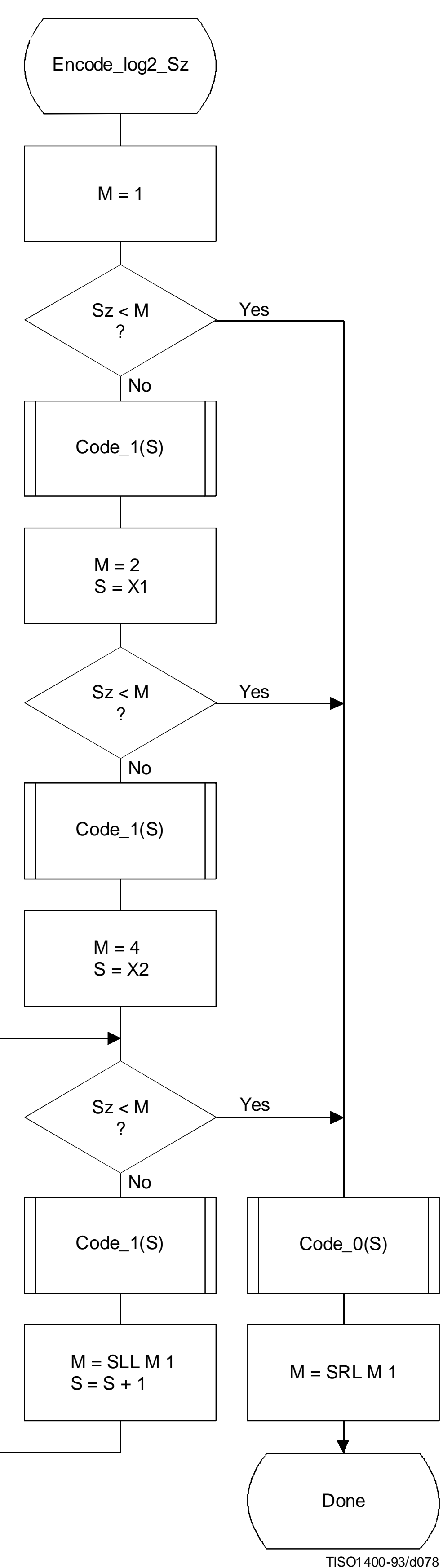


Figure F.8 – Decision sequence to establish the magnitude category

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.

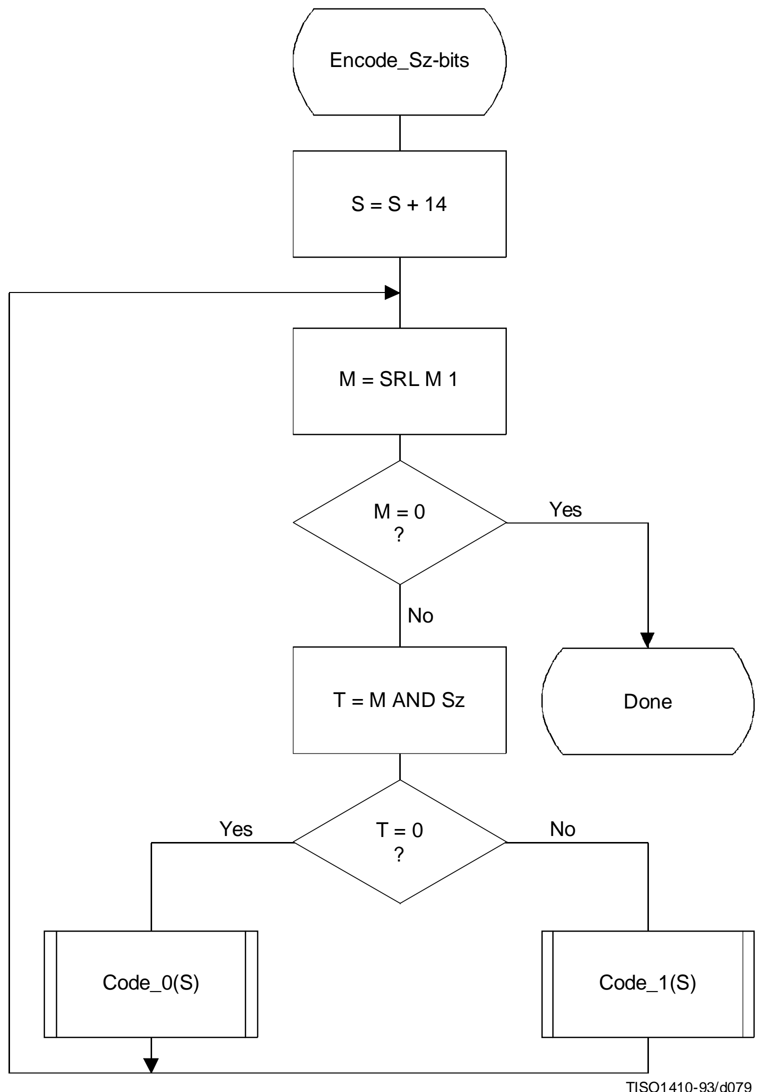


Figure F.9 – Decision sequence to code the magnitude bit pattern

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.

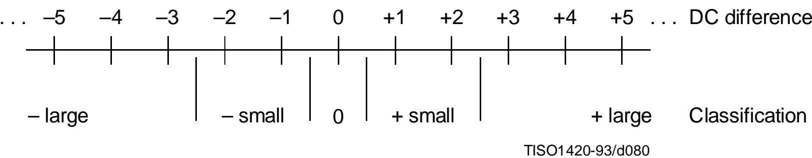


Figure F.10 – Conditioning classification of difference values

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.

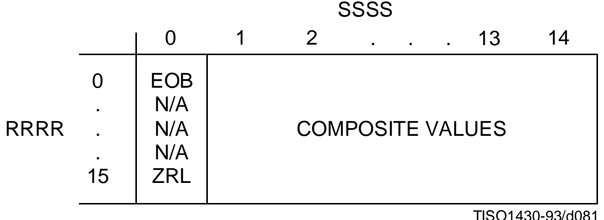


Figure F.11 – Two-dimensional value array for Huffman coding

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.

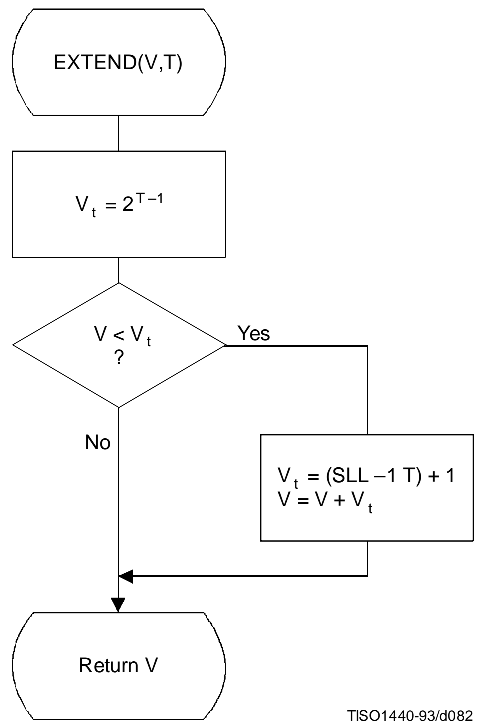


Figure F.12 – Extending the sign bit of a decoded value in V

F.2.2.2 Decoding procedure for AC coefficients

The decoding procedure for AC coefficients is shown in Figures F.13 and F.14.

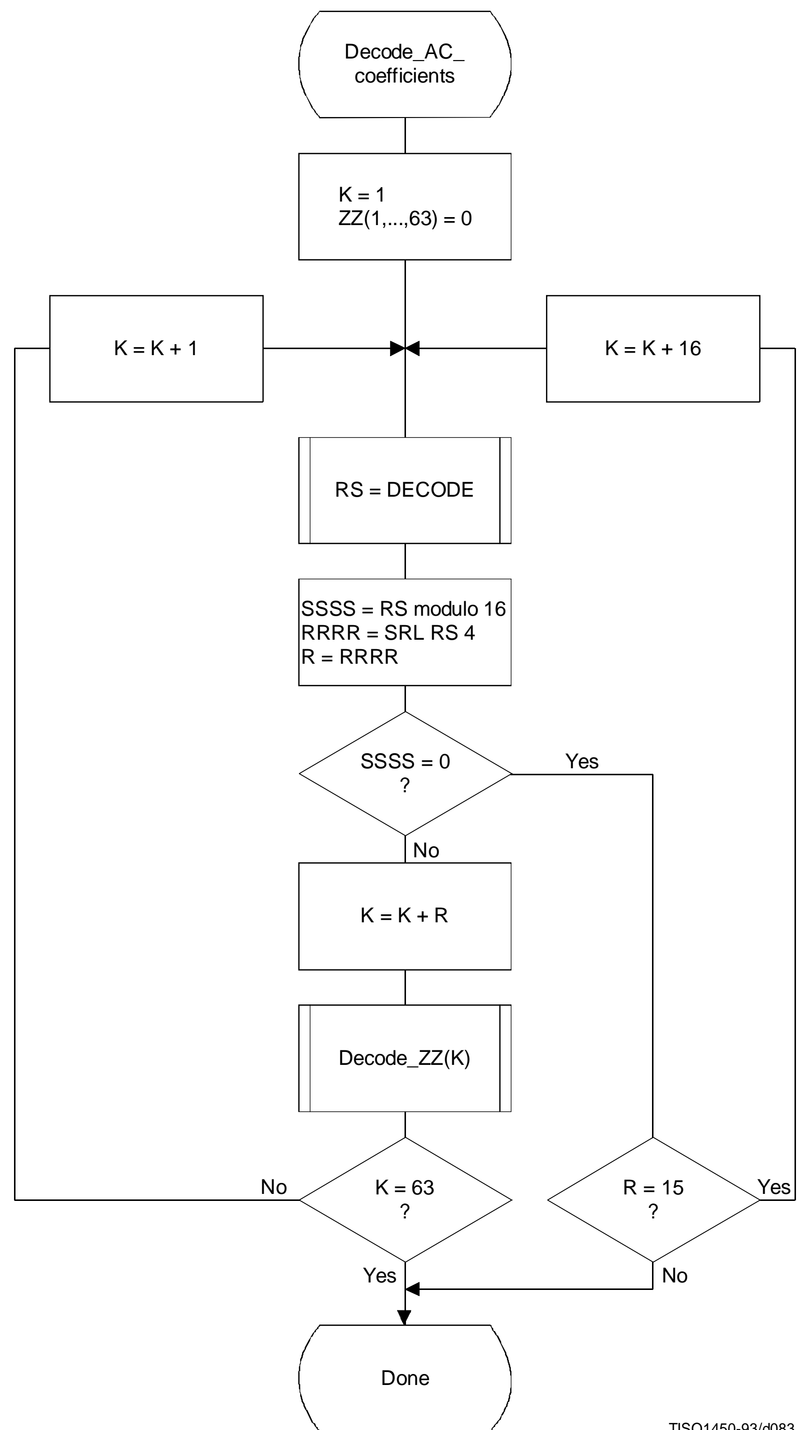


Figure F.13 – Huffman decoding procedure for AC coefficients

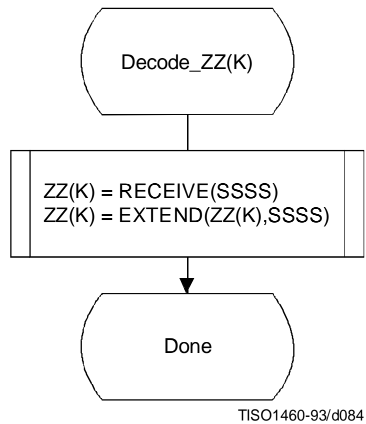


Figure F.14 – Decoding a non-zero AC coefficient

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.

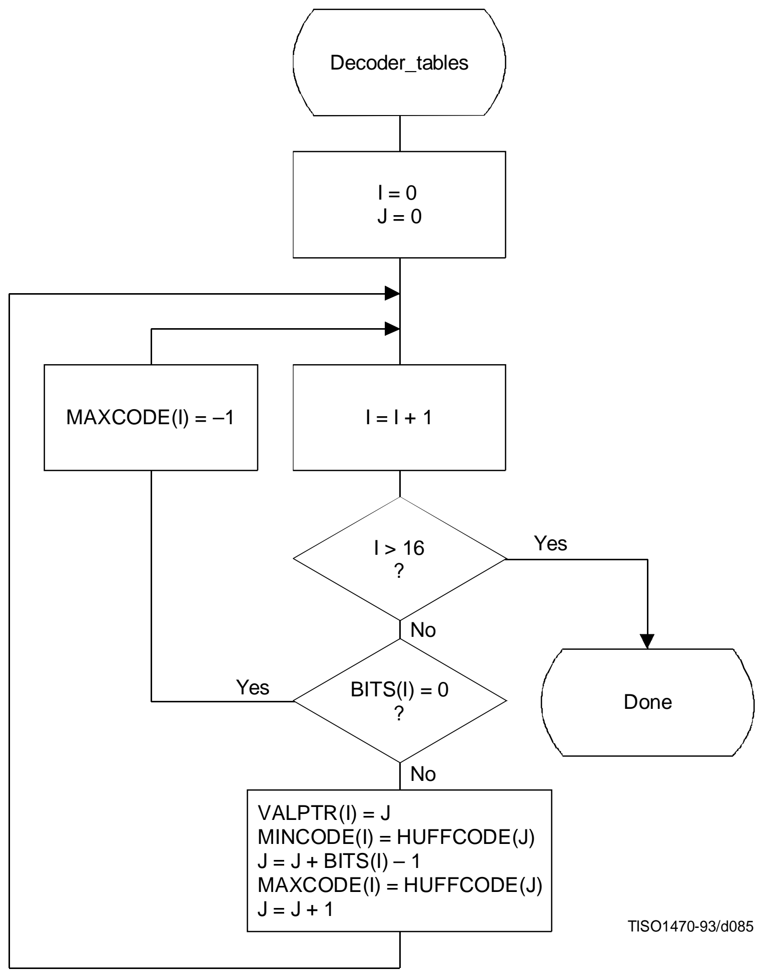


Figure F.15 – Decoder table generation

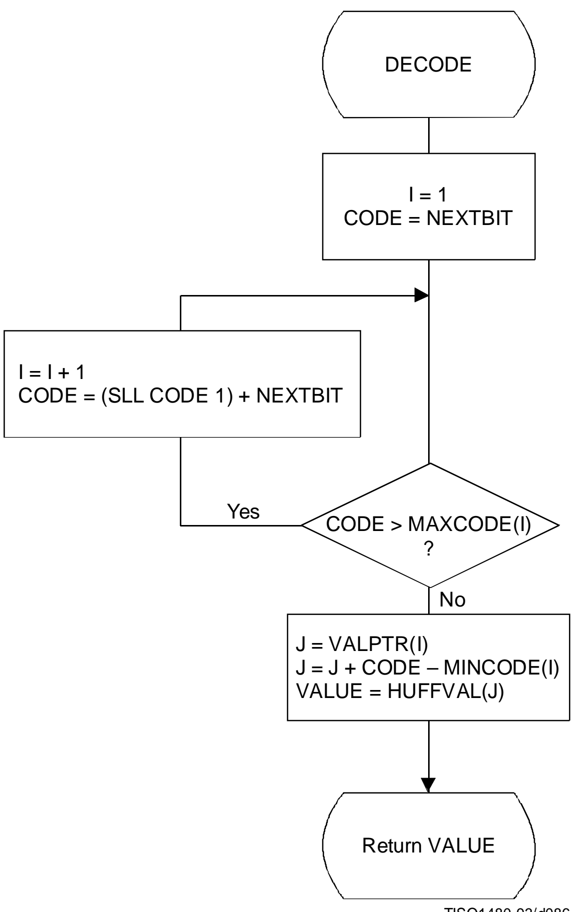


Figure F.16 – Procedure for DECODE

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).

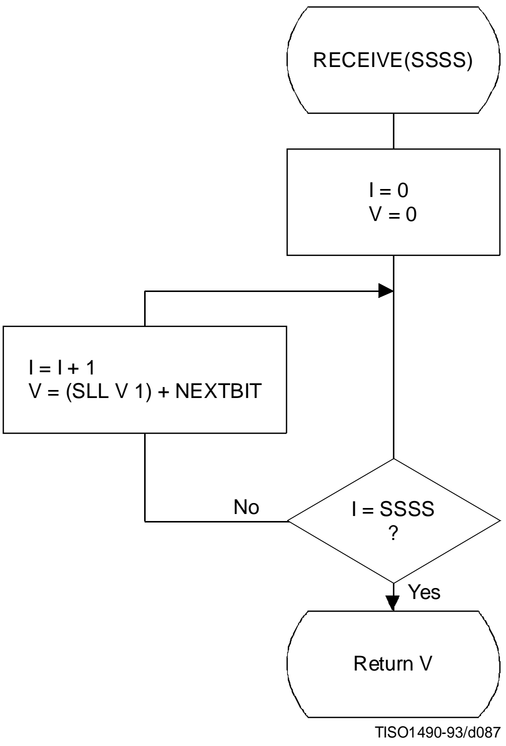


Figure F.17 – Procedure for RECEIVE(SSSS)

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.

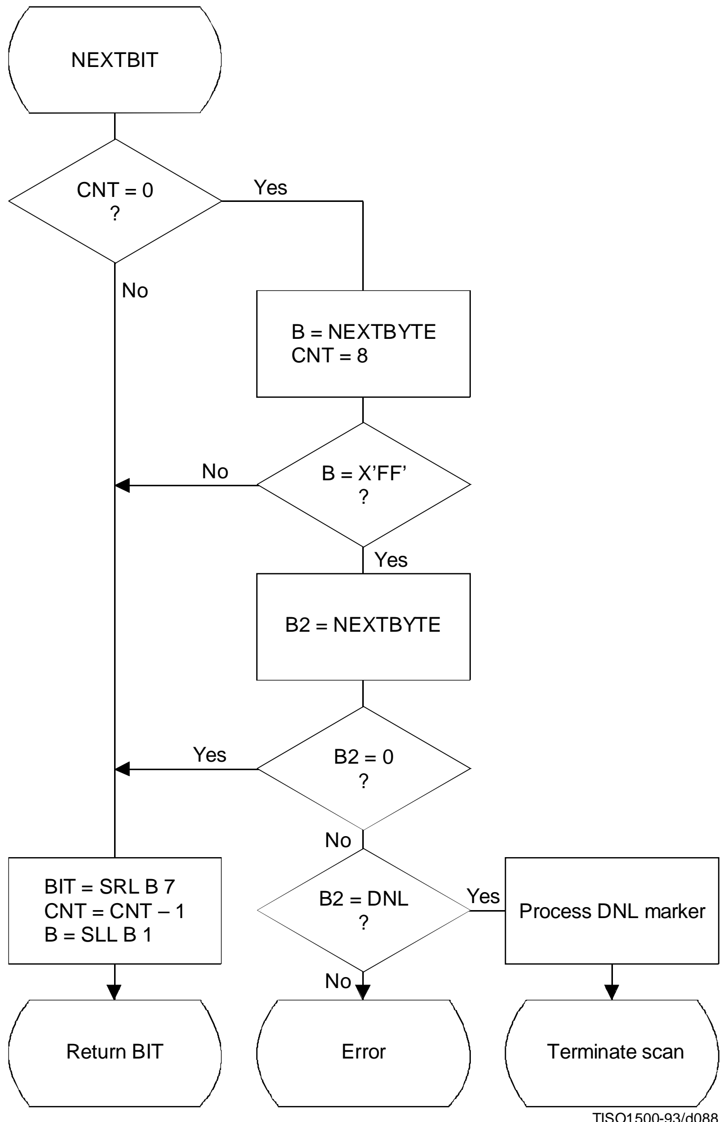


Figure F.18 – Procedure for fetching the next bit of compressed data

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.

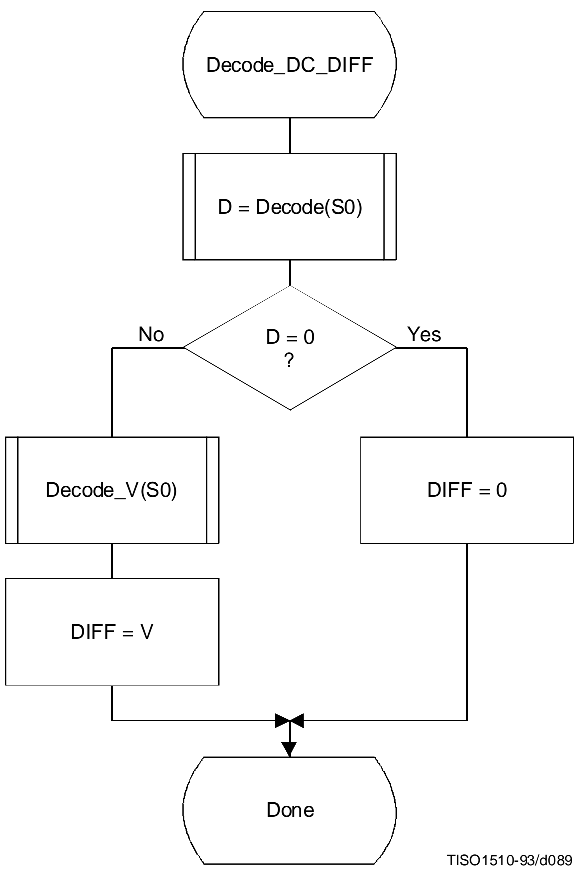


Figure F.19 – Arithmetic decoding of DC difference

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.

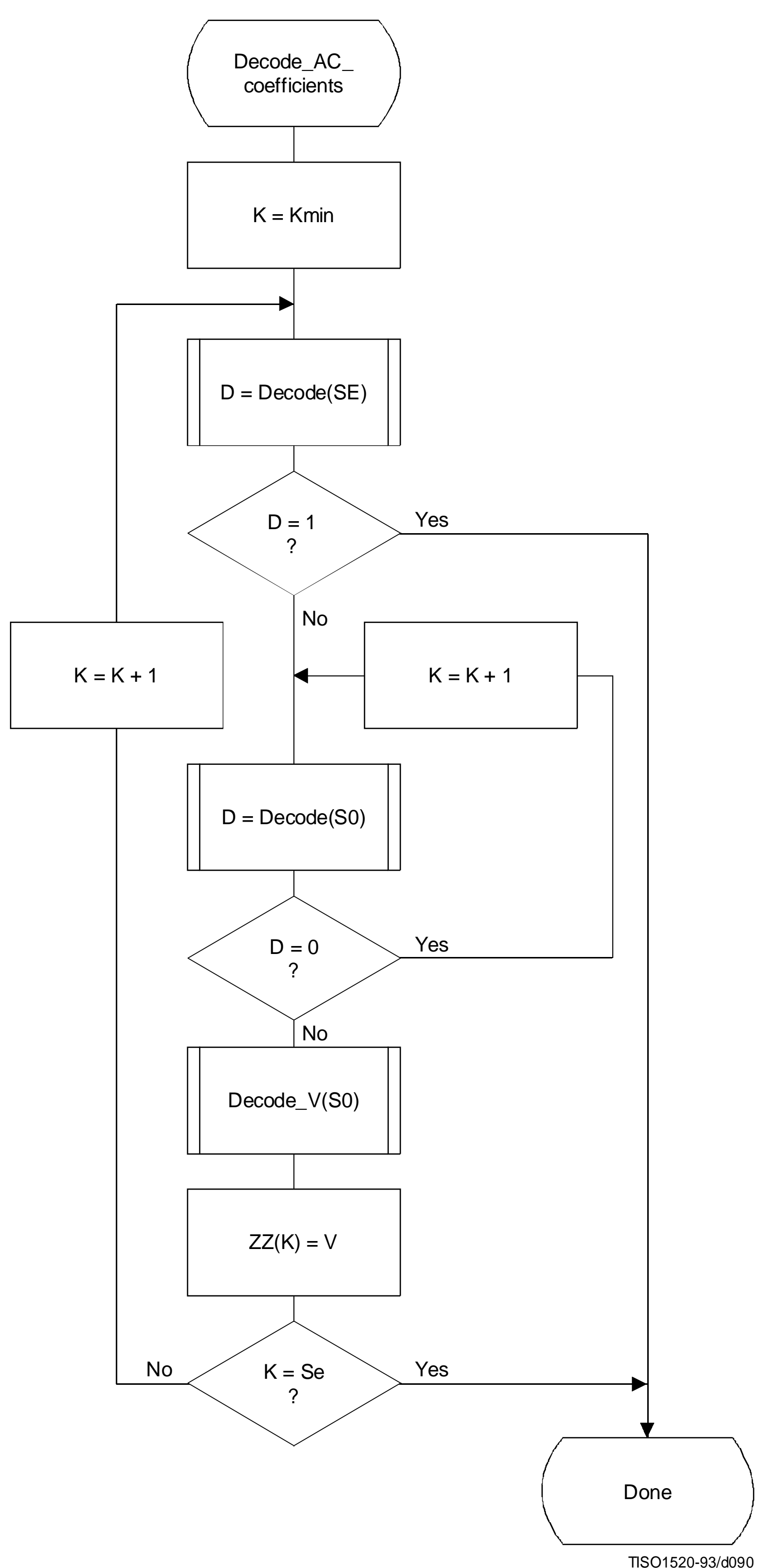


Figure F.20 – Procedure for decoding the AC coefficients

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.

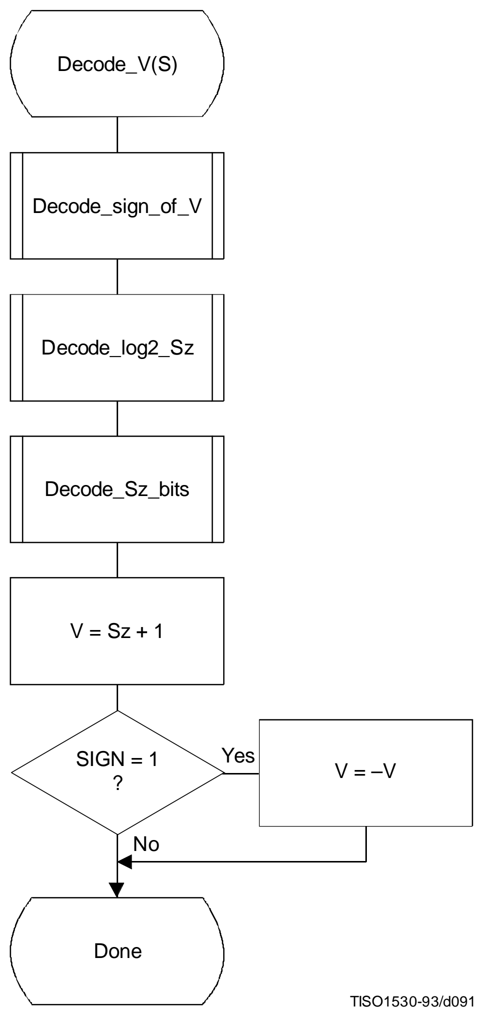


Figure F.21 – Sequence of procedures in decoding non-zero values of V

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.

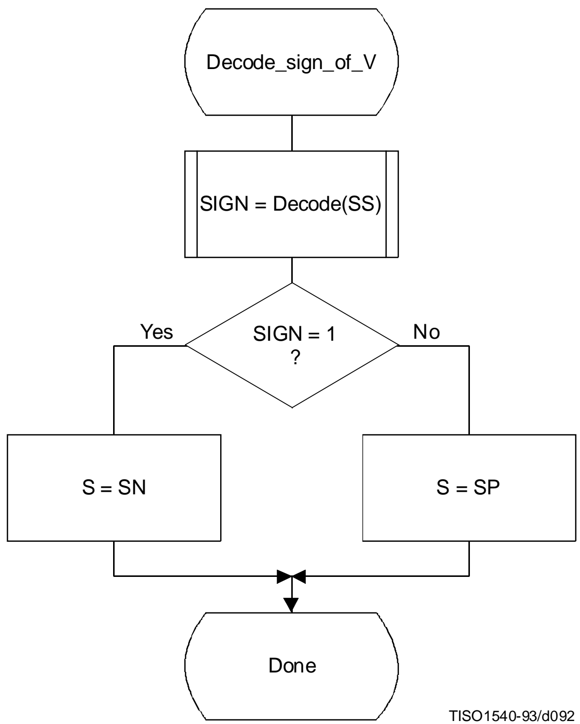


Figure F.22 – Decoding the sign of V

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.

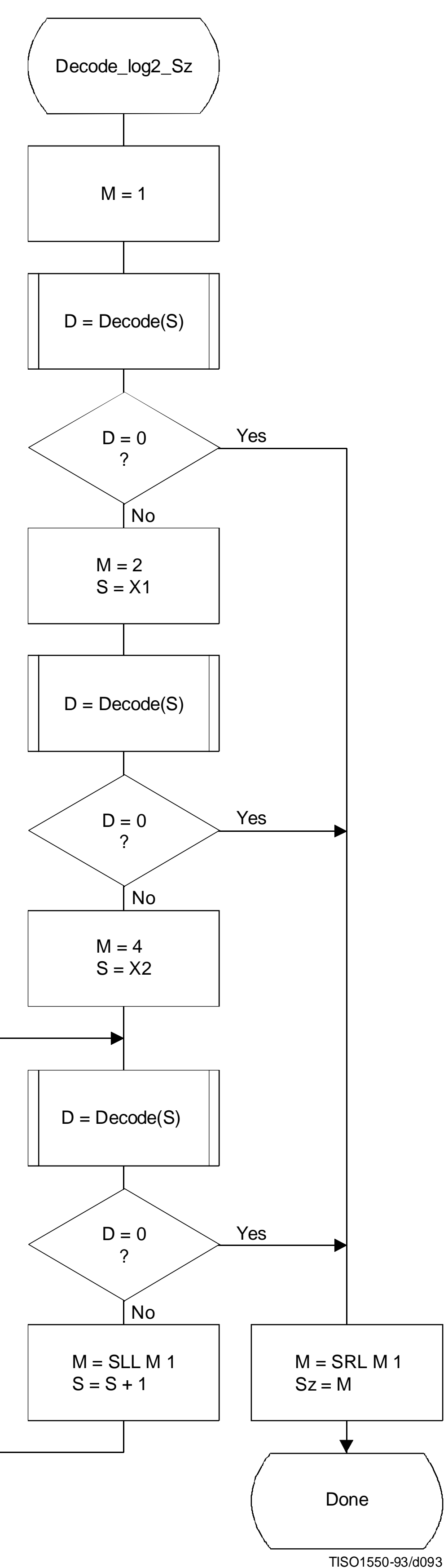


Figure F.23 – Decoding procedure to establish the magnitude category

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.

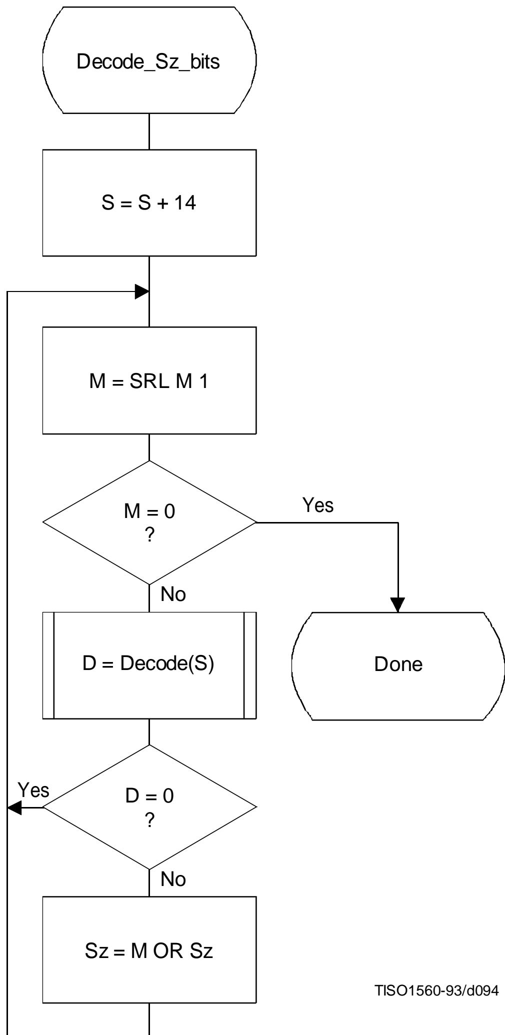


Figure F.24 – Decision sequence to decode the magnitude bit pattern

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.

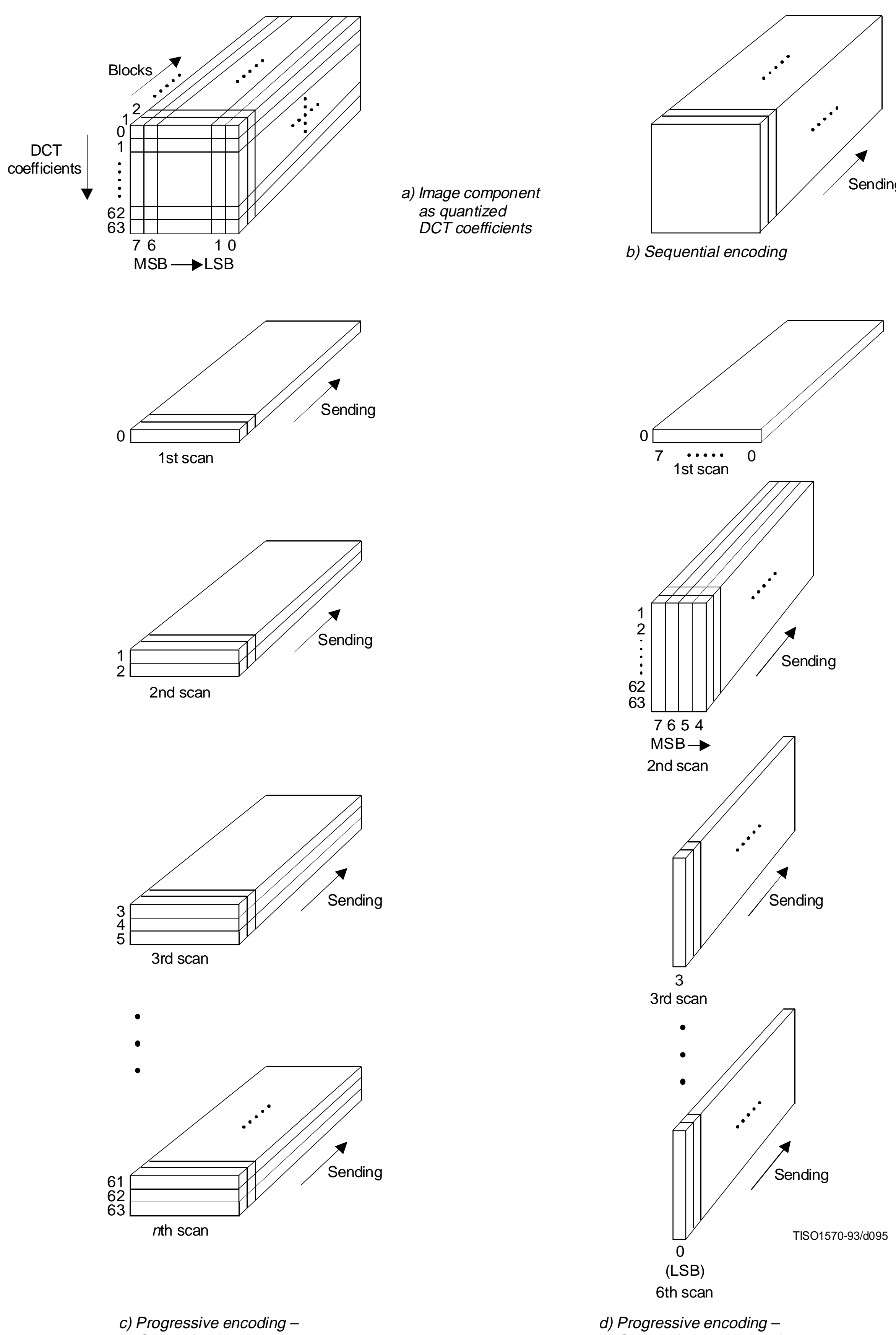


Figure G.1 – Spectral selection and successive approximation progressive processes

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.

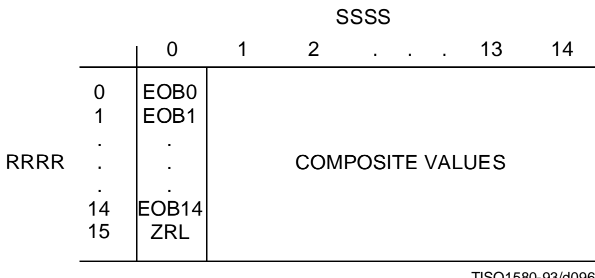


Figure G.2 – Two-dimensional value array for Huffman coding

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.

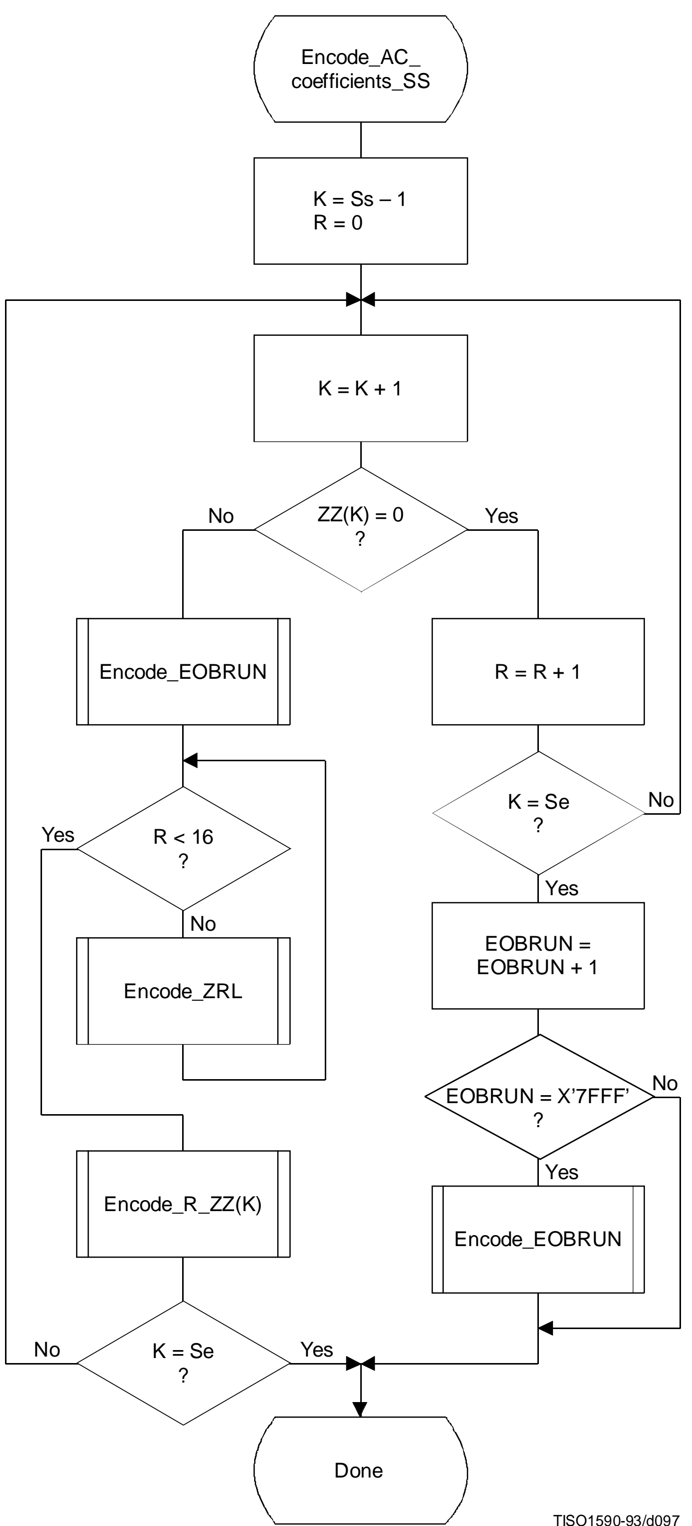


Figure G.3 – Procedure for progressive encoding of AC coefficients with Huffman coding

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).

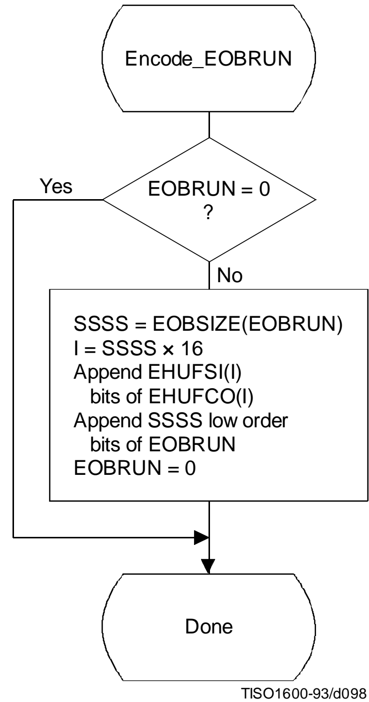


Figure G.4 – Progressive encoding of a non-zero AC coefficient

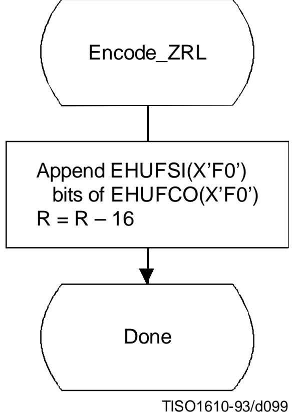


Figure G.5 – Encoding of the run of zero coefficients

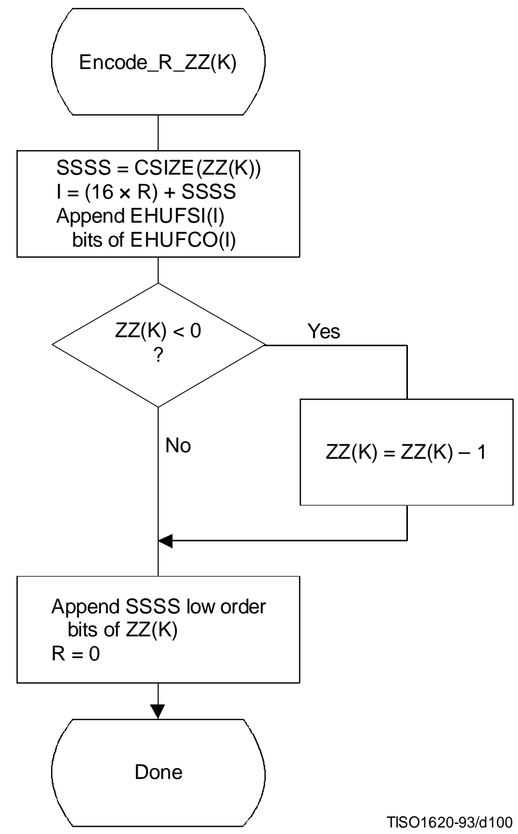


Figure G.6 – Encoding of the zero run and non-zero coefficient

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.

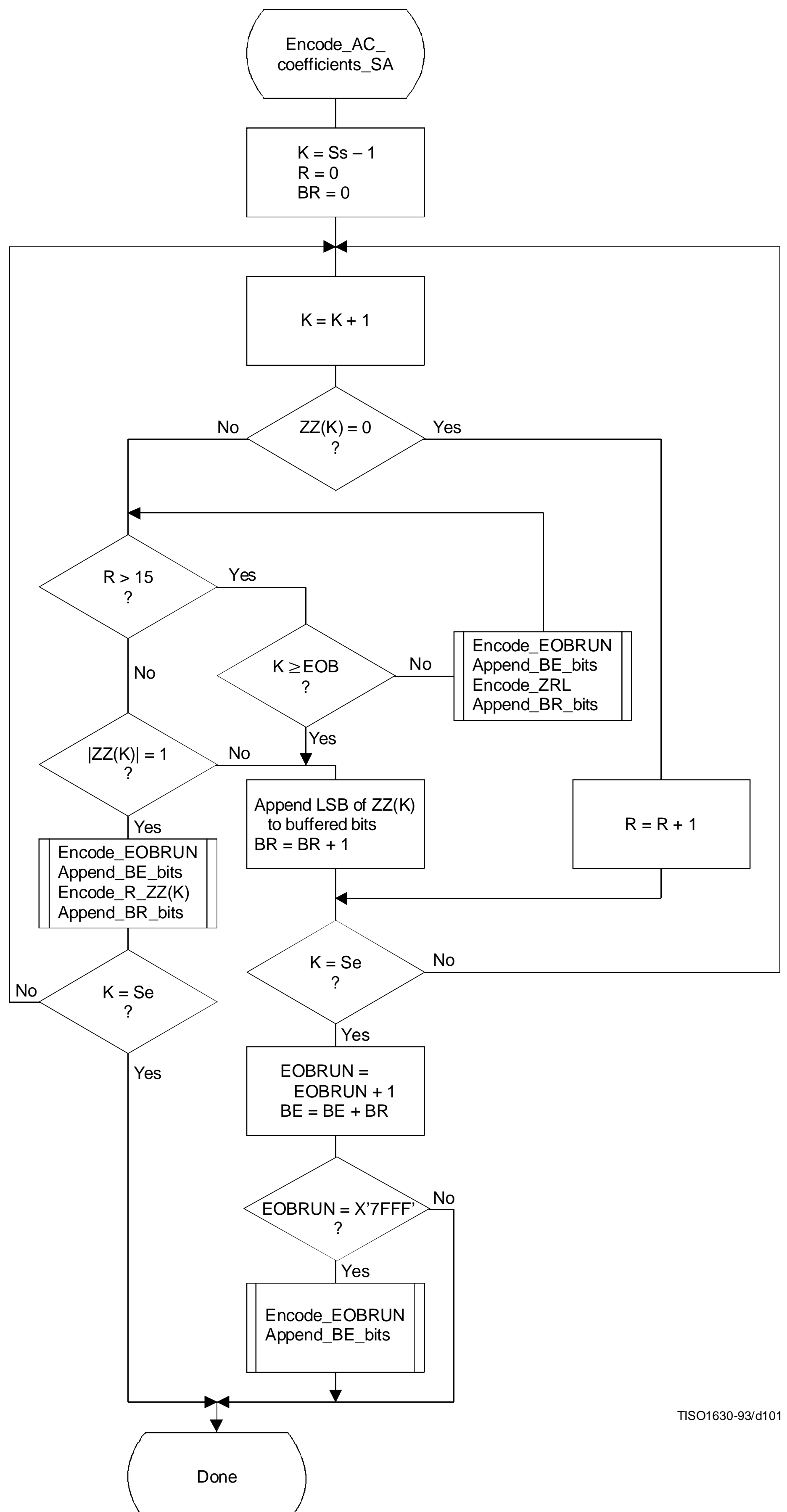


Figure G.7 – Successive approximation coding of AC coefficients using Huffman coding

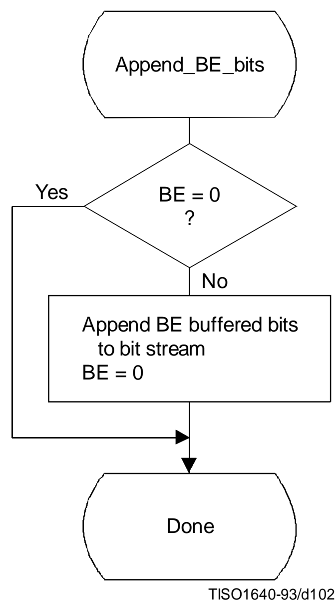


Figure G.8 – Transferring BE buffered bits from buffer to bit stream

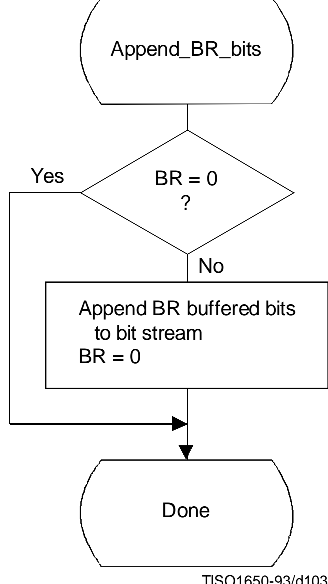


Figure G.9 – Transferring BR buffered bits from buffer to bit stream

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.

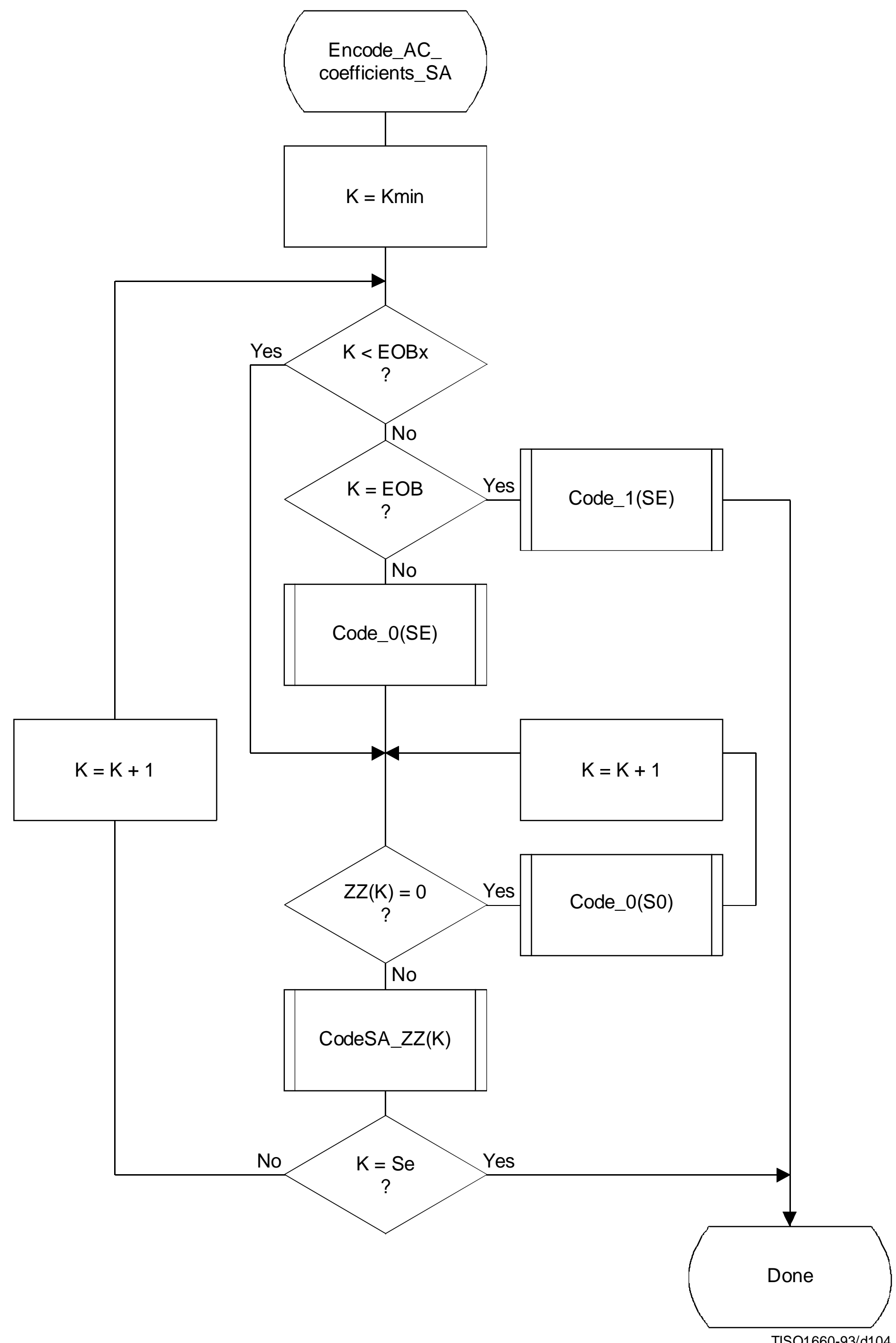


Figure G.10 – Subsequent successive approximation scans for coding of AC coefficients using arithmetic coding

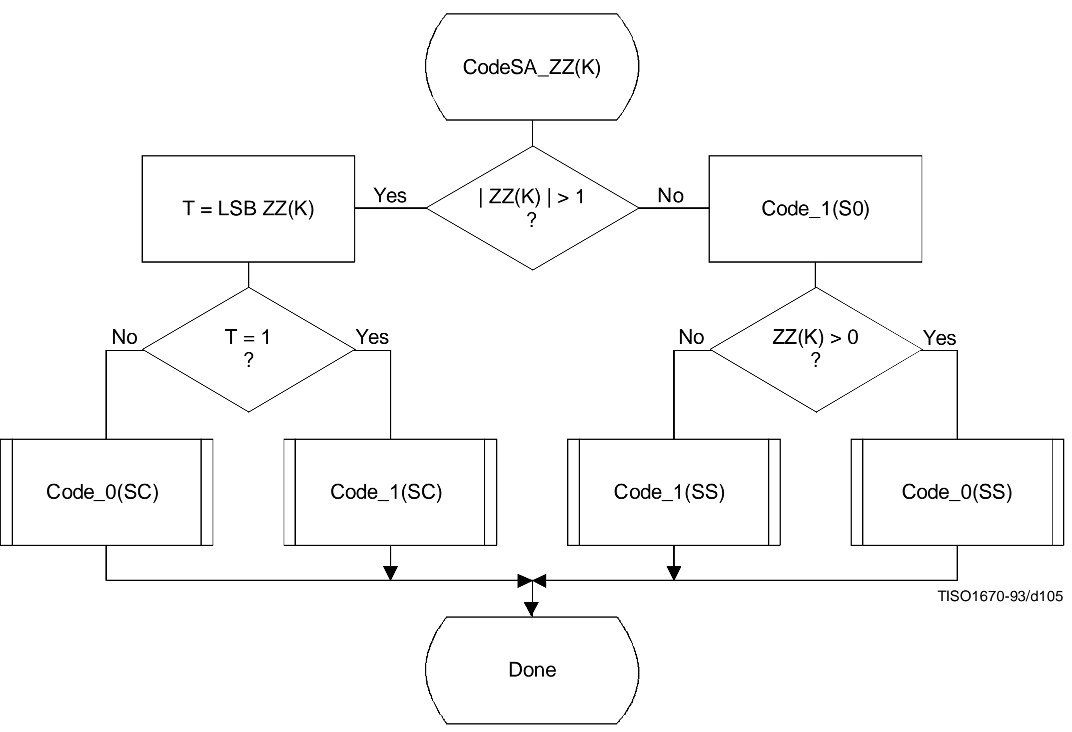


Figure G.11 – Coding non-zero coefficients for subsequent successive approximation scans

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.

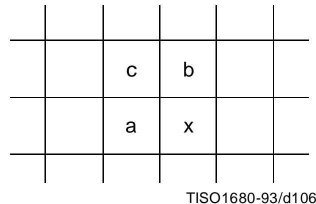


Figure H.1 – Relationship between sample and prediction samples

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.

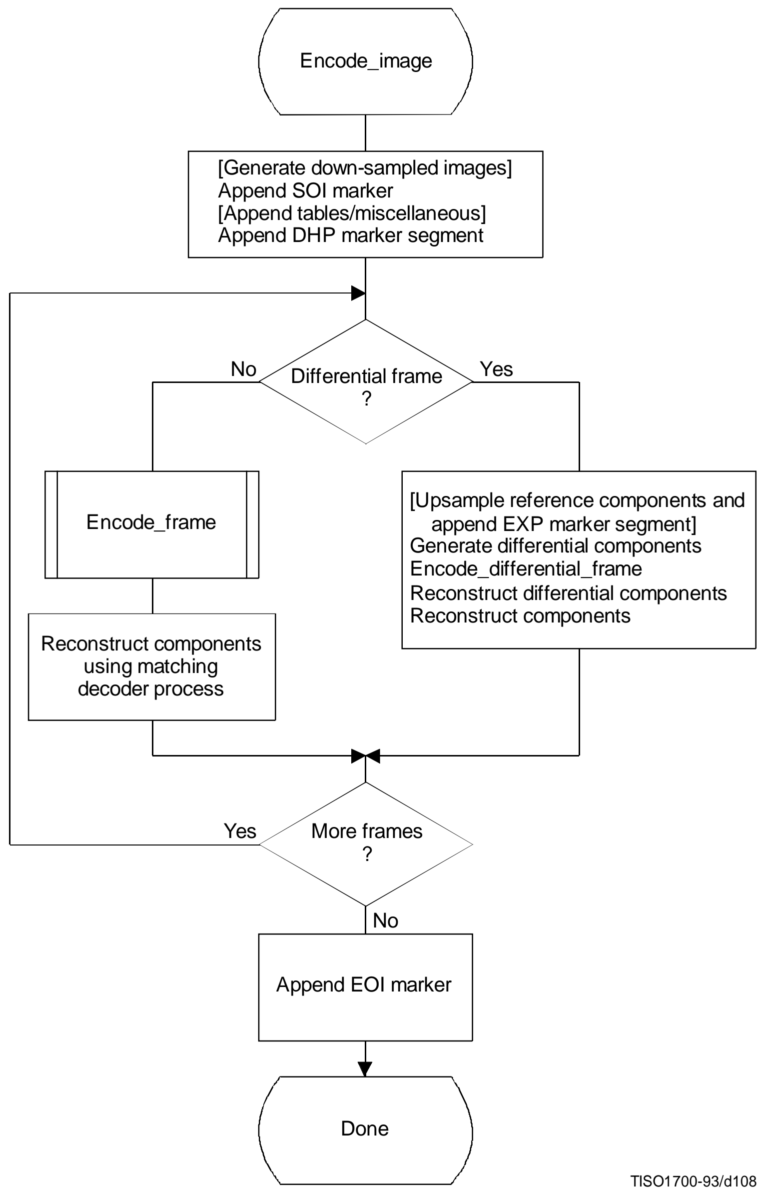


Figure J.1 – Hierarchical control procedure for encoding an image

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.

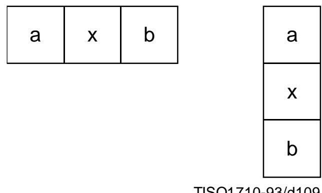


Figure J.2 – Diagram of sample positions for upsampling rules

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.

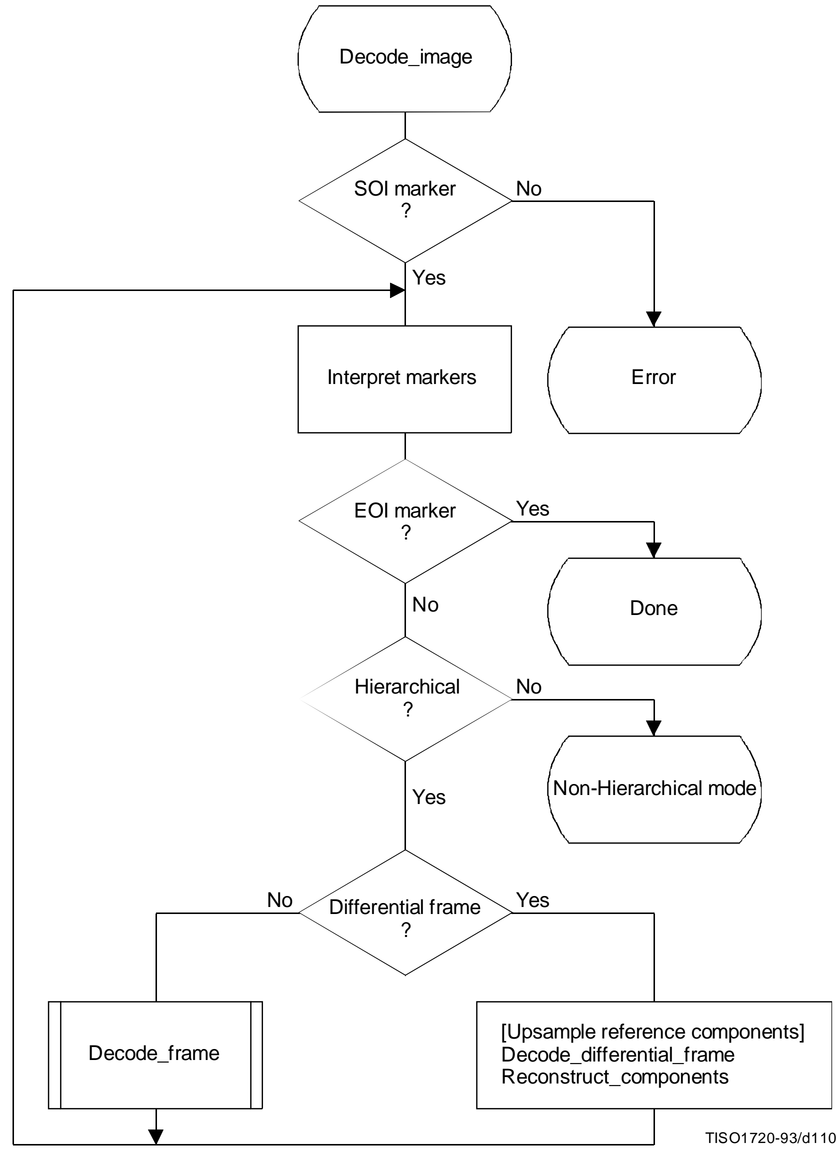


Figure J.3 – Hierarchical control procedure for decoding an image

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.

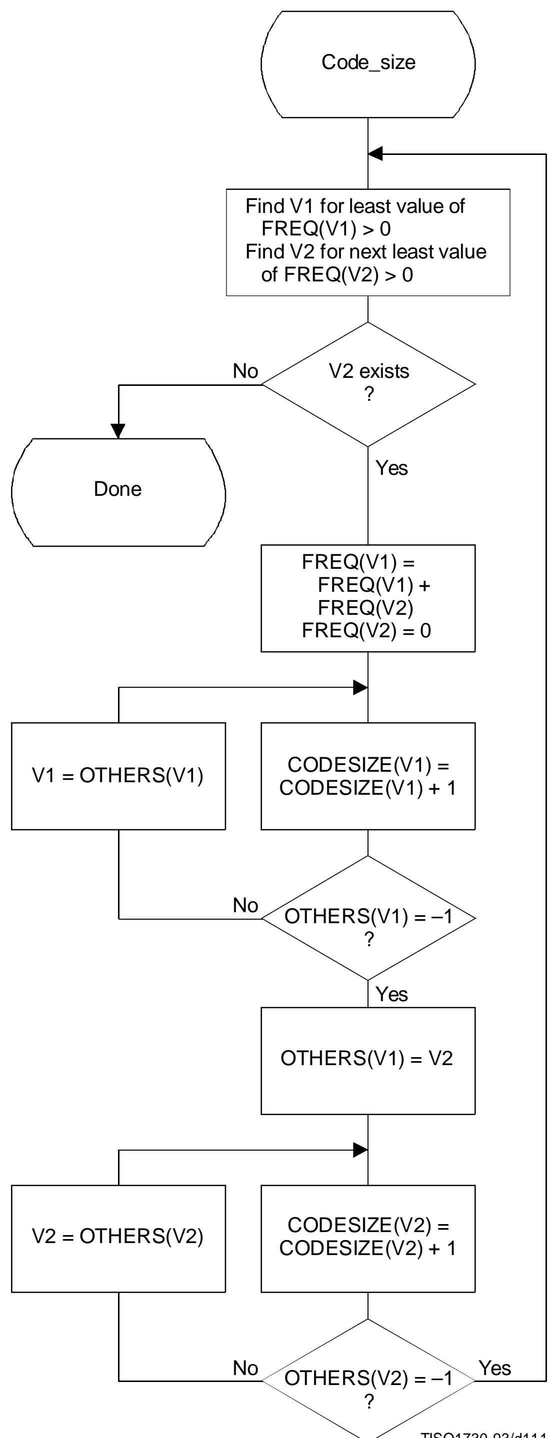


Figure K.1 – Procedure to find Huffman code sizes

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).

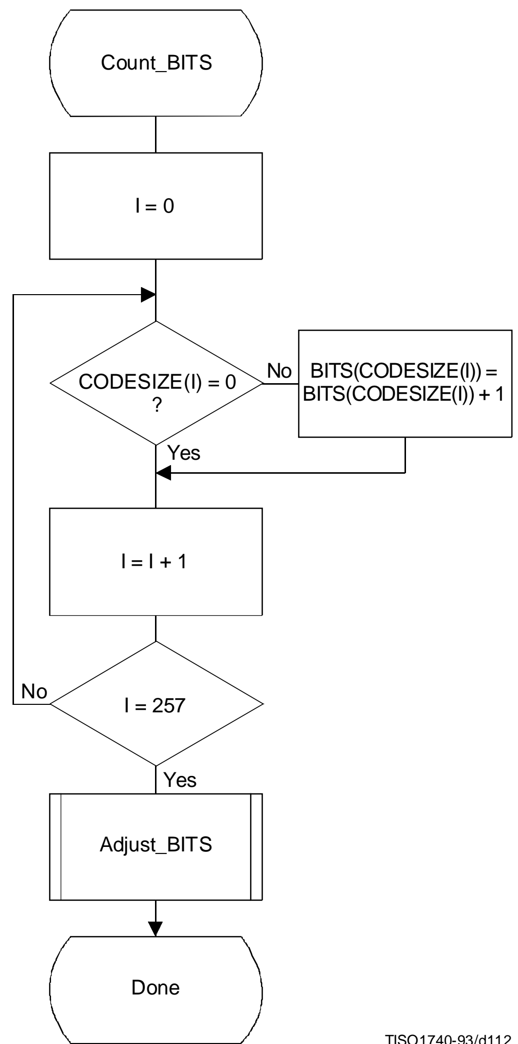


Figure K.2 – Procedure to find the number of codes of each size

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.

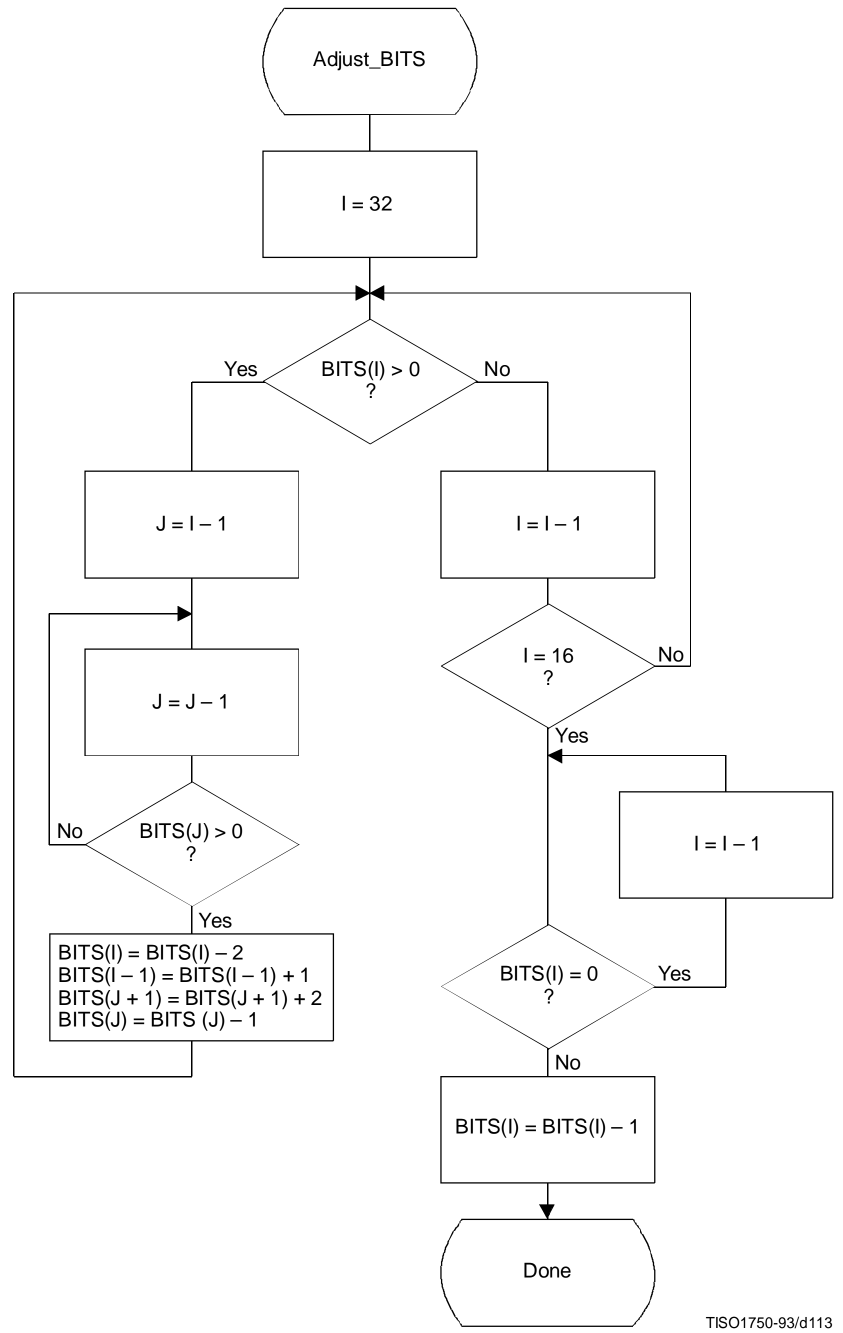


Figure K.3 – Procedure for limiting code lengths to 16 bits

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.

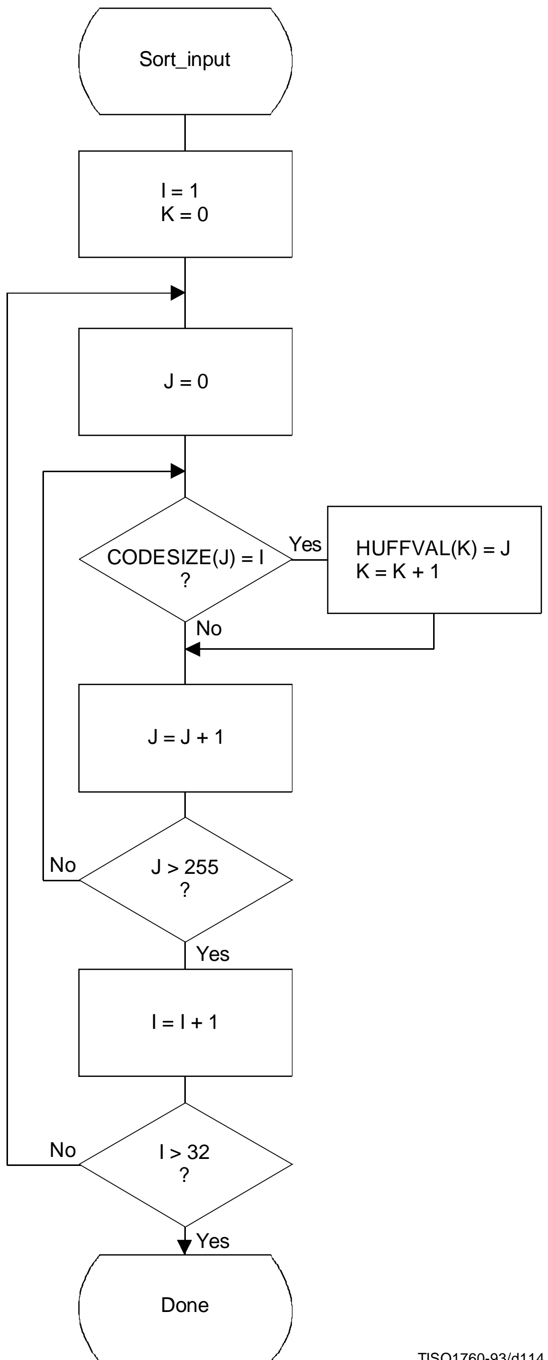


Figure K.4 – Sorting of input values according to code size

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.

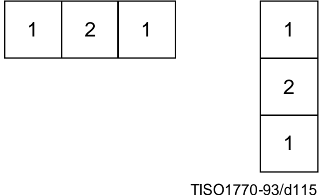


Figure K.5 – Low-pass filter example

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.

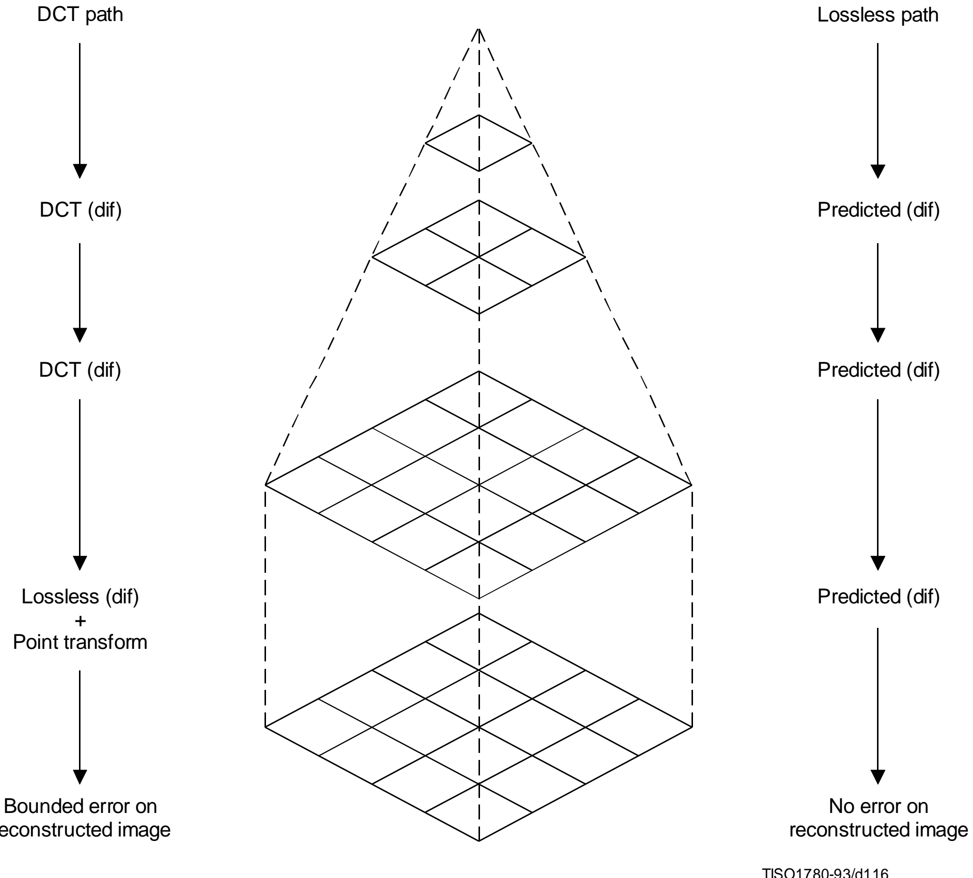


Figure K.6 – Sketch of the basic operations of the hierarchical mode

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.

```
DC  x   x   .   .   .   .   .
x   x   .   .   .   .   .   .
x   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .
```

TISO1790-93/d117

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

DC_1 DC_2 DC_3
DC_4 DC_5 DC_6
DC_7 DC_8 DC_9

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:

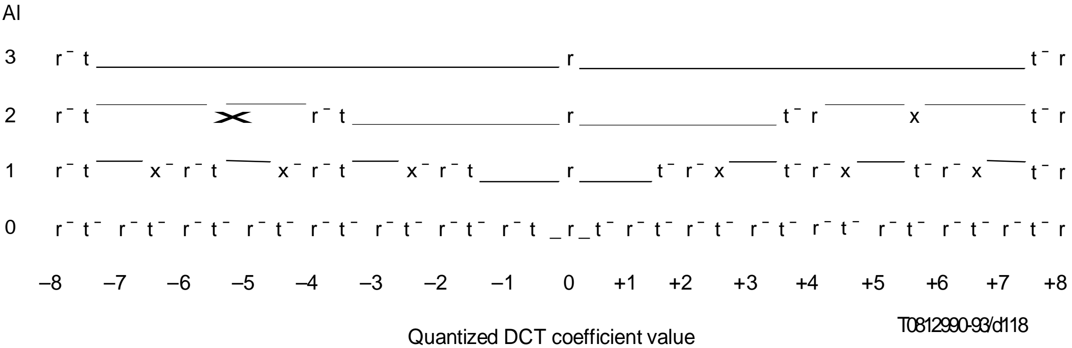


Figure K.8 – Illustration of two reconstruction strategies

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.

JPEG File Interchange Format
Version 1.02

September 1, 1992

Eric Hamilton
C-Cube Microsystems
1778 McCarthy Blvd.
Milpitas, CA 95035

+1 408 944-6300
Fax: +1 408 944-6314
E-mail: eric@c3.pla.ca.us

JPEG File Interchange Format
Version 1.02

Why a File Interchange Format

JPEG File Interchange Format is a minimal file format which enables JPEG bitstreams to be exchanged between a wide variety of platforms and applications. This minimal format does not include any of the advanced features found in the TIFF JPEG specification or any application specific file format. Nor should it, for the only purpose of this simplified format is to allow the exchange of JPEG compressed images.

JPEG File Interchange Format features

- Uses JPEG compression
- Uses JPEG interchange format compressed image representation
- PC or Mac or Unix workstation compatible
- Standard color space: one or three components. For three components, YCbCr (CCIR 601-256 levels)
- APP0 marker used to specify Units, X pixel density, Y pixel density, thumbnail
- APP0 marker also used to specify JFIF extensions
- APP0 marker also used to specify application-specific information

JPEG Compression

Although any JPEG process is supported by the syntax of the JPEG File Interchange Format (JFIF) it is strongly recommended that the JPEG baseline process be used for the purposes of file interchange. This ensures maximum compatibility with all applications supporting JPEG. JFIF conforms to the JPEG Draft International Standard (ISO DIS 10918-1).

The JPEG File Interchange Format is entirely compatible with the standard JPEG interchange format; the only additional requirement is the mandatory presence of the APP0 marker right after the SOI marker. Note that JPEG interchange format requires (as does JFIF) that all table specifications used in the encoding process be coded in the bitstream prior to their use.

Compatible across platforms

The JPEG File Interchange Format is compatible across platforms: for example, it does not

use any resource forks, supported by the Macintosh but not by PCs or workstations.

Standard color space

The color space to be used is YCbCr as defined by CCIR 601 (256 levels). The RGB components calculated by linear conversion from YCbCr shall not be gamma corrected (gamma = 1.0). If only one component is used, that component shall be Y.

APP0 marker used to identify JPEG FIF

The APP0 marker is used to identify a JPEG FIF file. The JPEG FIF APP0 marker is mandatory right after the SOI marker.

The JFIF APP0 marker is identified by a zero terminated string: "JFIF". The APP0 can be used for any other purpose by the application provided it can be distinguished from the JFIF APP0.

The JFIF APP0 marker provides information which is missing from the JPEG stream: version number, X and Y pixel density (dots per inch or dots per cm), pixel aspect ratio (derived from X and Y pixel density), thumbnail.

APP0 marker used to specify JFIF extensions

Additional APP0 marker segment(s) can optionally be used to specify JFIF extensions. If used, these segment(s) must immediately follow the JFIF APP0 marker. Decoders should skip any unsupported JFIF extension segments and continue decoding.

The JFIF extension APP0 marker is identified by a zero terminated string: "JFXX". The JFIF extension APP0 marker segment contains a 1-byte code which identifies the extension. This version, version 1.02, has only one extension defined: an extension for defining thumbnails stored in formats other than 24-bit RGB.

APP0 marker used for application-specific information

Additional APP0 marker segments can be used to hold application-specific information which does not affect the decodability or displayability of the JFIF file. Application-specific APP0 marker segments must appear after the JFIF APP0 and any JFXX APP0 segments. Decoders should skip any unrecognized application-specific APP0 segments.

Application-specific APP0 marker segments are identified by a zero terminated string which identifies the application (not "JFIF" or "JFXX"). This string should be an organization name or company trademark. Generic strings such as dog, cat, tree, etc. should not be used.

Conversion to and from RGB

Y, Cb, and Cr are converted from R, G, and B as defined in CCIR Recommendation 601 but are normalized so as to occupy the full 256 levels of a 8-bit binary encoding. More precisely:

```
Y   = 256 * E'y
Cb  = 256 * [ E'Cb ] + 128
Cr  = 256 * [ E'Cr ] + 128
```

where the E'y, E'Cb and E'Cb are defined as in CCIR 601. Since values of E'y have a range of 0 to 1.0 and those for E'Cb and E'Cr have a range of -0.5 to +0.5, Y, Cb, and Cr must be clamped to 255 when they are maximum value.

RGB to YCbCr Conversion

YCbCr (256 levels) can be computed directly from 8-bit RGB as follows:

```
Y   =     0.299  R + 0.587  G + 0.114  B
Cb  =   - 0.1687 R - 0.3313 G + 0.5    B + 128
Cr  =     0.5    R - 0.4187 G - 0.0813 B + 128
```

NOTE - Not all image file formats store image samples in the order R0, G0, B0, ... Rn, Gn, Bn. Be sure to verify the sample order before converting an RGB file to JFIF.

YCbCr to RGB Conversion

RGB can be computed directly from YCbCr (256 levels) as follows:

```
R = Y                    + 1.402   (Cr-128)
G = Y - 0.34414 (Cb-128) - 0.71414 (Cr-128)
B = Y + 1.772   (Cb-128)
```

Image Orientation

In JFIF files, the image orientation is always top-down. This means that the first image samples encoded in a JFIF file are located in the upper left hand corner of the image and encoding proceeds from left to right and top to bottom. Top-down orientation is used for both the full resolution image and the thumbnail image.

The process of converting an image file having bottom-up orientation to JFIF must include inverting the order of all image lines before JPEG encoding

Spatial Relationship of Components

Specification of the spatial positioning of pixel samples within components relative to the samples of other components is necessary for proper image post processing and accurate image presentation. In JFIF files, the position of the pixels in subsampled components are defined with respect to the highest resolution component. Since components must be sampled orthogonally (along rows and columns), the spatial position of the samples in a given subsampled component may be determined by specifying the horizontal and vertical offsets of the first sample, i.e. the sample in the upper left corner, with respect to the highest resolution component.

The horizontal and vertical offsets of the first sample in a subsampled component, Xoffseti[0,0] and Yoffseti[0,0], is defined to be

```
Xoffseti[0,0] = ( Nsamplesref / Nsamplesi  ) / 2 - 0.5
Yoffseti[0,0] = ( Nlinesref / Nlinesi  ) / 2 - 0.5
```

where
Nsamplesref is the number of samples per line in the largest component,
Nsamplesi is the number of samples per line in the ith component,
Nlinesref is the number of lines in the largest component,
Nlinesi is the number of lines in the ith component.

Proper subsampling of components incorporates an anti-aliasing filter which reduces the spectral bandwidth of the full resolution components. Subsampling can easily be accomplished using a symmetrical digital filter with an even number of taps (coefficients). A commonly used filter for 2:1 subsampling utilizes two taps (1/2,1/2).

NOTE - This definition is compatible with industry standards such as Postcript Level 2 and QuickTime. This defintition is not compatible with the conventions used by CCIR Recommendation 601-1 and other digital video formats. For these formats, pre-processing of the chrominance components is necessary prior to compression in order to ensure accurate reconstruction of the compressed image.

JPEG File Interchange Format Specification

The syntax of a JFIF file conforms to the syntax for interchange format defined in Annex B of ISO DIS 10918-1. In addition, a JFIF file uses APP0 marker segments and constrains certain parameters in the frame header as defined below.

```
    X'FF', SOI
         X'FF', APP0, length, identifier, version, units, Xdensity, Ydensity, Xthumbnail,
          Ythumbnail, (RGB)n
              length     (2 bytes)  Total APP0 field byte count, including the byte
                                    count value (2 bytes), but excluding the APP0
                                    marker itself
              identifier (5 bytes)  = X'4A', X'46', X'49', X'46', X'00'
                                    This zero terminated string ("JFIF") uniquely
                                    identifies this APP0 marker.  This string shall
                                    have zero parity (bit 7=0).
              version    (2 bytes)  = X'0102'
```

```
                                The most significant byte is used for major
                                revisions, the least significant byte for minor
                                revisions. Version 1.02 is the current released
                                revision.
              units      (1 byte)   Units for the X and Y densities.
                                units = 0:  no units, X and Y specify the pixel
                                     aspect ratio
                                units = 1:  X and Y are dots per inch
                                units = 2:  X and Y are dots per cm
              Xdensity   (2 bytes)  Horizontal pixel density
              Ydensity   (2 bytes)  Vertical pixel density
              Xthumbnail (1 byte)   Thumbnail horizontal pixel count
              Ythumbnail (1 byte)   Thumbnail vertical pixel count
              (RGB)n     (3n bytes) Packed (24-bit) RGB values for the thumbnail
                                pixels, n = Xthumbnail * Ythumbnail
        [ Optional JFIF extension APP0 marker segment(s) - see below ]
                o
                o
                o
        X'FF', SOFn, length, frame parameters
           Number of components Nf  = 1 or 3
           1st component    C1    = 1 = Y component
           2nd component    C2    = 2 = Cb component
           3rd component    C3    = 3 = Cr component
                o
                o
                o
    X'FF', EOI
```

JFIF Extension APP0 Marker Segment

Immediately following the JFIF APP0 marker segment may be a JFIF extension APP0 marker. This JFIF extension APP0 marker segment may only be present for JFIF versions 1.02 and above. The syntax of the JFIF extension APP0 marker segment is:

```
         X'FF', APP0, length, identifier, extension_code, extension_data
            length   (2 bytes)    Total APP0 field byte count, including the byte
                                count value (2 bytes), but excluding the APP0
                                marker itself
            identifier  (5 bytes)    = X'4A', X'46', X'58', X'58', X'00'
                                This zero terminated string ("JFXX") uniquely
                                identifies this APP0 marker.  This string shall
                                have zero parity (bit 7=0).
            extension_code (1 byte)    = Code which identifies the extension.  In this
                                version, the following extensions are defined:
                                   = X'10'   Thumbnail coded using JPEG
                                   = X'11'   Thumbnail stored using 1 byte/pixel
                                   = X'13'   Thumbnail stored using 3 bytes/pixel
            extension_data (variable)    = The specification of the remainder of the JFIF
                                extension APP0 marker segment varies with the
                                extension. See below for a specification of
                                extension_data for each extension.
```

JFIF Extension: Thumbnail coded using JPEG

This extension supports thumbnails compressed using JPEG. The compressed thumbnail immediately follows the extension_code (X'10') in the extension_data field and the length of the compressed data must be included in the JFIF extension APP0 marker length field.

The syntax of the extension_data field conforms to the syntax for interchange format defined in Annex B of ISO DIS 10918-1. However, no "JFIF" or "JFXX" marker segments shall be present. As in the full resolution image of the JFIF file, the syntax of extension_data constrains parameters in the frame header as defined below:

```
    X'FF', SOI
                o
                o
                o
        X'FF', SOFn, length, frame parameters
             Number of components    Nf    = 1 or 3
             1st component    C1    = 1 = Y component
             2nd component    C2    = 2 = Cb component
             3rd component    C3    = 3 = Cr component
```

```
                o
                o
                o
    X'FF', EOI
```

JFIF Extension: Thumbnail stored using one byte per pixel

This extension supports thumbnails stored using one byte per pixel and a color palette in the extension_data field. The syntax of extension_data is:

```
        Xthumbnail       (1 byte)    Thumbnail horizontal pixel count
        Ythumbnail       (1 byte)    Thumbnail vertical pixel count
        palette          (768 bytes)  24-bit RGB pixel values for the color palette.
                                      The RGB values define the colors represented by
                                      each value of an 8-bit binary encoding (0 - 255).
        (pixel)n        (n bytes)     8-bit values for the thumbnail pixels
                                      n = Xthumbnail * Ythumbnail
```

JFIF Extension: Thumbnail stored using three bytes per pixel

This extension supports thumbnails stored using three bytes per pixel in the extension_data field. The syntax of extension_data is:

```
        Xthumbnail       (1 byte)    Thumbnail horizontal pixel count
        Ythumbnail       (1 byte)    Thumbnail vertical pixel count
        (RGB)n           (3n bytes)  Packed (24-bit) RGB values for the thumbnail
                                     pixels, n = Xthumbnail * Ythumbnail
```

Useful tips

o you can identify a JFIF file by looking for the following sequence: X'FF', SOI, X'FF', APP0, <2 bytes to be skipped>, "JFIF", X'00'.

o if you use APP0 elsewhere, be sure not to have the strings "JFIF" or "JFXX" right after the APP0 marker.

o if you do not want to include a thumbnail, just program Xthumbnail = Ythumbnail = 0.

o be sure to check the version number in the special APP0 field. In general, if the major version number of the JFIF file matches that supported by the decoder, the file will be decodable.

o if you only want to specify a pixel aspect ratio, put 0 for the units field in the special APP0 field. Xdensity and Ydensity can then be programmed for the desired aspect ratio. Xdensity = 1, Ydensity = 1 will program a 1:1 aspect ratio. Xdensity and Ydensity should always be non-zero.

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

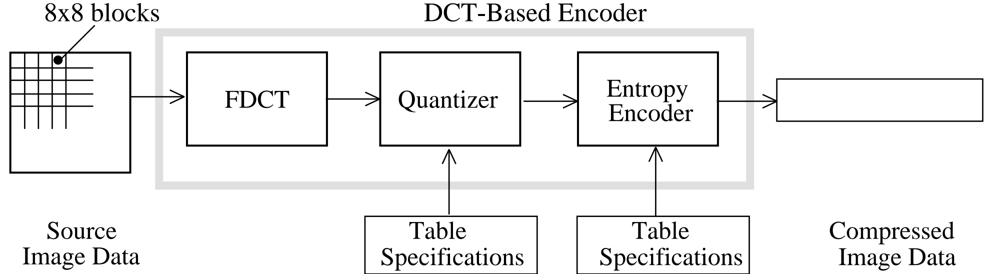


Figure 1. DCT-Based Encoder Processing Steps

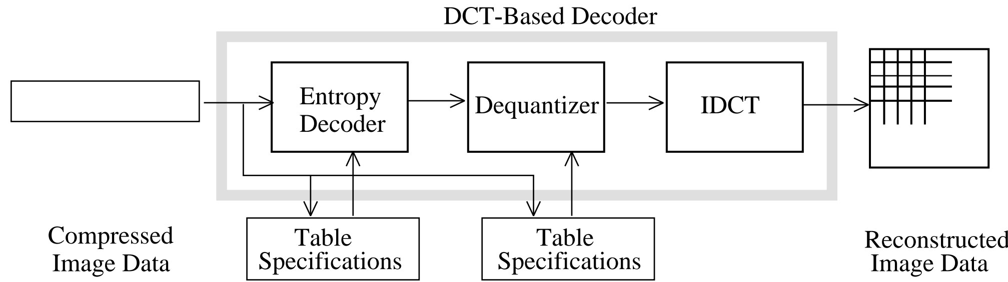


Figure 2. DCT-Based Decoder Processing Steps

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:

$$F^Q(u,v) = \mathit{Integer\ Round}\left(\frac{F(u,v)}{Q(u,v)}\right) \quad (3)$$

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:

$$F^{Q'}(u,v) = F^Q(u,v) * Q(u,v) \quad (4)$$

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.

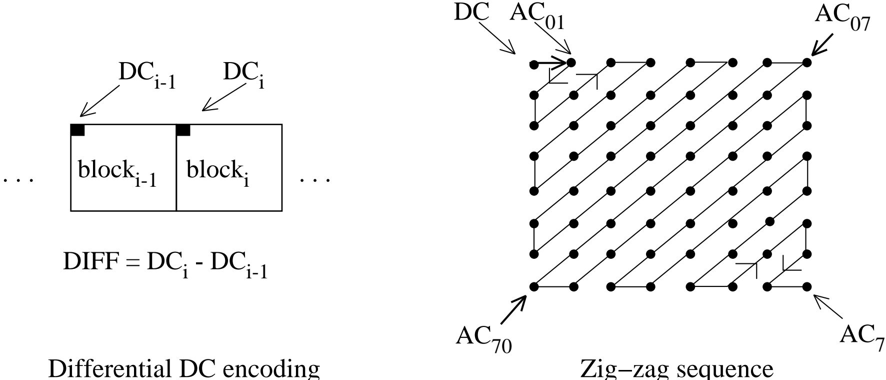


Figure 3. Preparation of Quantized Coefficients for Entropy Coding

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

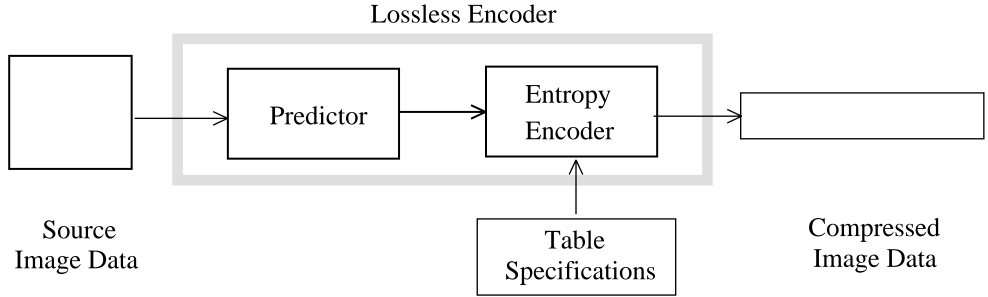


Figure 4. Lossless Mode Encoder Processing Steps

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).

	C	B		
	A	X		

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

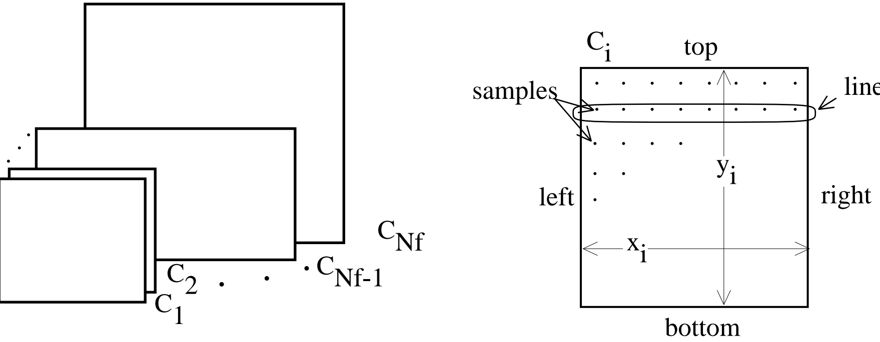
Table 1. Predictors for Lossless Coding

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.

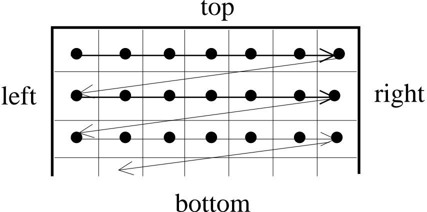


Figure 7. Noninterleaved Data Ordering

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

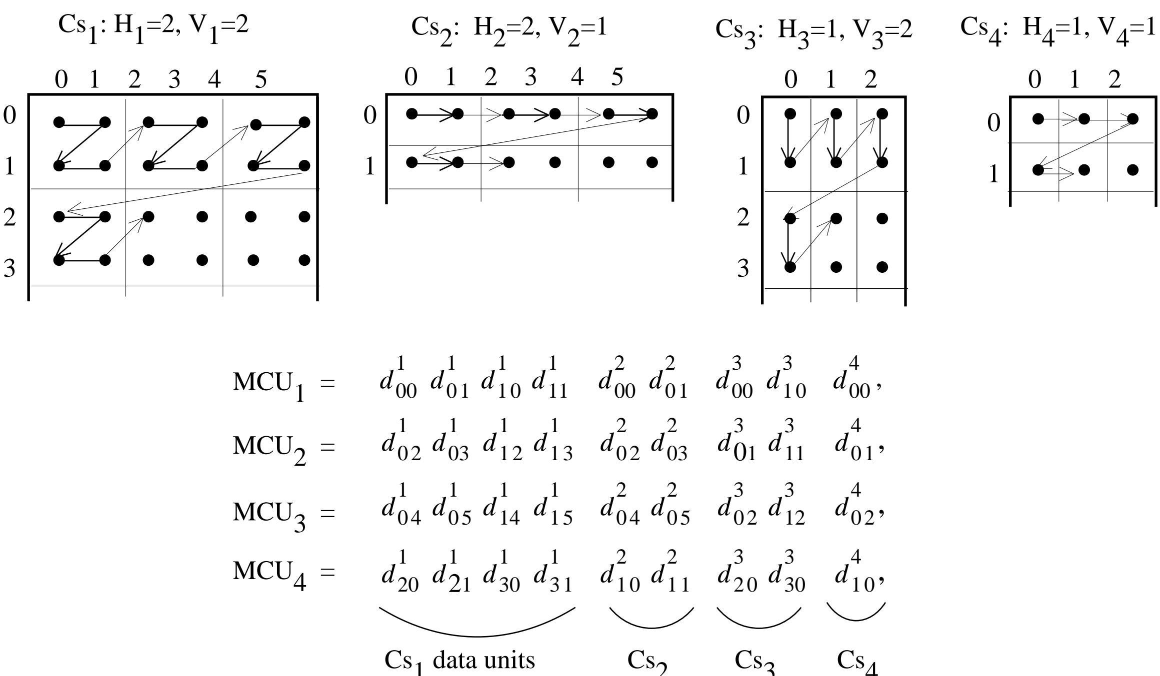


Figure 8. Generalized Interleaved Data Ordering Example

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:

$$\sum_{\substack{\textit{all } \text{i in} \\ \text{interleave}}} H_i \times V_i \leq 10 \qquad (6)$$

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.)

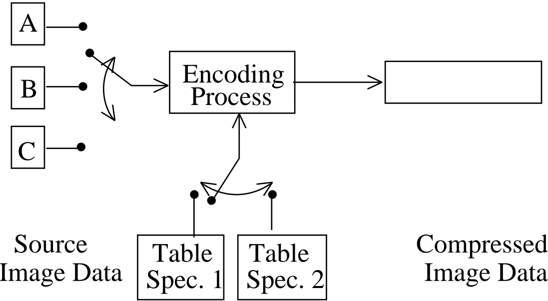


Figure 9. Component-Interleave and Table-Switching Control

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		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.

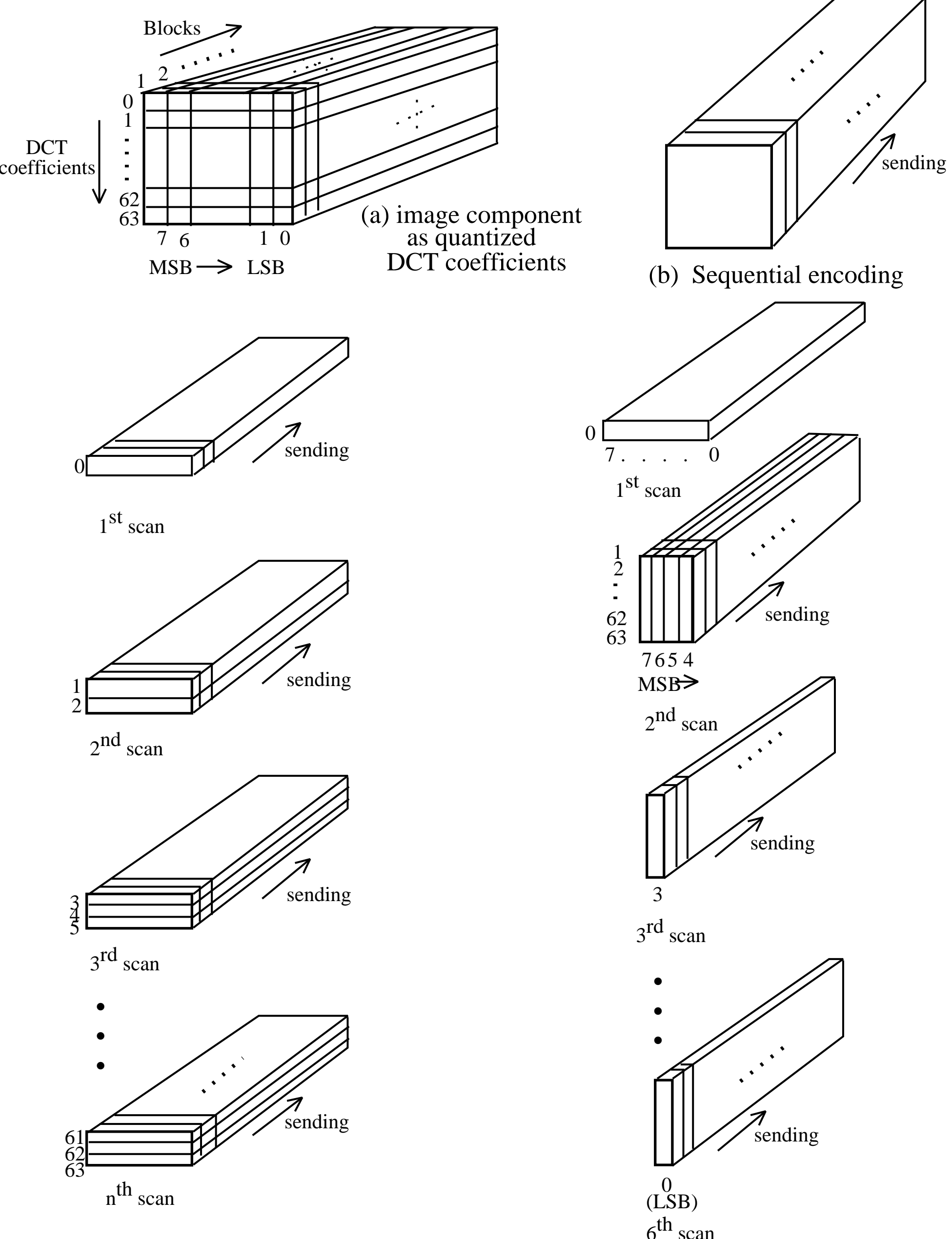


Figure 11. Spectral Selection and Successive Approximation Methods of Progressive Encoding

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.

LZW and GIF explained----Steve Blackstock

I hope this little document will help enlighten those of you out there who want to know more about the Lempel-Ziv Welch compression algorithm, and, specifically, the implementation that GIF uses.

Before we start, here's a little terminology, for the purposes of this document:

"character": a fundamental data element. In normal text files, this is just a single byte. In raster images, which is what we're interested in, it's an index that specifies the color of a given pixel. I'll refer to an arbitray character as "K".

"charstream": a stream of characters, as in a data file.

"string": a number of continuous characters, anywhere from one to very many characters in length. I can specify an arbitrary string as "[...]K".

"prefix": almost the same as a string, but with the implication that a prefix immediately precedes a character, and a prefix can have a length of zero. So, a prefix and a character make up a string. I will refer to an arbitrary prefix as "[...]".

"root": a single-character string. For most purposes, this is a character, but we may occasionally make a distinction. It is [...]K, where [...] is empty.

"code": a number, specified by a known number of bits, which maps to a string.

"codestream": the output stream of codes, as in the "raster data"

"entry": a code and its string.

"string table": a list of entries; usually, but not necessarily, unique.

That should be enough of that.

LZW is a way of compressing data that takes advantage of repetition of strings in the data. Since raster data usually contains a lot of this repetition, LZW is a good way of compressing and decompressing it.

For the moment, lets consider normal LZW encoding and decoding. GIF's variation on the concept is just an extension from there.

LZW manipulates three objects in both compression and decompression: the charstream, the codestream, and the string table. In compression, the charstream is the input and the codestream is the output. In decompression, the codestream is the input and the charstream is the output. The string table is a product of both compression and decompression, but is never passed from one to the other.

The first thing we do in LZW compression is initialize our string table. To do this, we need to choose a code size (how many bits) and know how many values our characters can possibly take. Let's say our code size is 12 bits, meaning we can store 0->FFF, or 4096 entries in our string table. Lets also say that we have 32 possible different characters. (This corresponds to, say, a picture in which there are 32 different colors possible for each pixel.) To initialize the table, we set code#0 to character#0, code #1 to character#1, and so on, until code#31 to character#31. Actually, we are specifying that each code from 0 to 31 maps to a root. There will be no more entries in the table that have this property.

Now we start compressing data. Let's first define something called the "current prefix". It's just a prefix that we'll store things in and compare things to now and then. I will refer to it as "[.c.]". Initially, the current prefix has nothing in it. Let's also define a "current string", which will be the current prefix plus the next character in the charstream. I will refer to the current string as "[.c.]K", where K is some character. OK, look at the first character in the charstream. Call it P. Make [.c.]P the current string. (At this point, of course, it's just the root P.) Now search through the string table to see if [.c.]P appears in it. Of course, it does now, because our string table is initialized to have all roots. So we don't do anything. Now make [.c.]P the current prefix. Look at the next character in the charstream. Call it Q. Add it to the current prefix to form [.c.]Q, the current string. Now search through the string table to see if [.c.]Q appears in it. In this case, of course, it doesn't. Aha! Now we get to do something. Add [.c.]Q (which is PQ in this case) to the string table for code#32, and output the code for [.c.] to the codestream. Now start over again with the current prefix being just the root P. Keep adding characters to [.c.] to form [.c.]K, until you can't find [.c.]K in the string table. Then output the code for [.c.] and add [.c.]K to the string table. In pseudo-code, the algorithm goes something like this:

```
[1] Initialize string table;
```

```
[2] [.c.] <- empty;
[3] K <- next character in charstream;
[4] Is [.c.]K in string table?
 (yes: [.c.] <- [.c.]K;
       go to [3];
 )
 (no: add [.c.]K to the string table;
      output the code for [.c.] to the codestream;
      [.c.] <- K;
      go to [3];
 )
```

It's as simple as that! Of course, when you get to step [3] and there aren't any more characters left, you just output the code for [.c.] and throw the table away. You're done.

Wanna do an example? Let's pretend we have a four-character alphabet: A,B,C,D. The charstream looks like ABACABA. Let's compress it. First, we initialize our string table to: #0=A, #1=B, #2=C, #3=D. The first character is A, which is in the string table, so [.c.] becomes A. Next we get AB, which is not in the table, so we output code #0 (for [.c.]),
and add AB to the string table as code #4. [.c.] becomes B. Next we get [.c.]A = BA, which is not in the string table, so output code #1, and add BA to the string table as code #5. [.c.] becomes A. Next we get AC, which is not in the string table. Output code #0, and add AC to the string table as code #6. Now [.c.] becomes C. Next we get [.c.]A = CA, which is not in the table. Output #2 for C, and add CA to table as code#7. Now [.c.] becomes A. Next we get AB, which IS in the string table, so [.c.] gets AB, and we look at ABA, which is not in the string table, so output the code for AB, which is #4, and add ABA to the string table as code #8. [.c.] becomes A. We can't get any more characters, so we just output #0 for the code for A, and we're done. So, the codestream is #0#1#0#2#4#0.

A few words (four) should be said here about efficiency: use a hashing strategy. The search through the string table can be computationally intensive, and some hashing is well worth the effort. Also, note that "straight LZW" compression runs the risk of overflowing the string table - getting to a code which can't be represented in the number of bits you've set aside for codes. There are several ways of dealing with this problem, and GIF implements a very clever one, but we'll get to that.

An important thing to notice is that, at any point during the compression, if [...]K is in the string table, [...] is there also. This fact suggests an efficient method for storing strings in the table. Rather than store the entire string of K's in the table, realize that any string can be expressed as a prefix plus a character: [...]K. If we're about to store [...]K in the table, we know that [...] is already there, so we can just store the code for [...] plus the final character K.

Ok, that takes care of compression. Decompression is perhaps more difficult conceptually, but it is really easier to program.

Here's how it goes: We again have to start with an initialized string table. This table comes from what knowledge we have about the charstream that we will eventually get, like what possible values the characters can take. In GIF files, this information is in the header as the number of possible pixel values. The beauty of LZW, though, is that this is all we need to know. We will build the rest of the string table as we decompress the codestream. The compression is done in such a way that we will never encounter a code in the codestream that we can't translate into a string.

We need to define something called a "current code", which I will refer to as "<code>", and an "old-code", which I will refer to as "<old>". To start things off, look at the first code. This is now <code>. This code will be in the intialized string table as the code for a root. Output the root to the charstream. Make this code the old-code <old>. *Now look at the next code, and make it <code>. It is possible that this code will not be in the string table, but let's assume for now that it is. Output the string corresponding to <code> to the codestream. Now find the first character in the string you just translated. Call this K. Add this to the prefix [...] generated by <old> to form a new string [...]K. Add this string [...]K to the string table, and set the old-code <old> to the current code <code>. Repeat from where I typed the asterisk, and you're all set. Read this paragraph again if you just skimmed it!!! Now let's consider the possibility that <code> is not in the string table. Think back to compression, and try to understand what happens when you have a string like P[...]P[...]PQ appear in the charstream. Suppose P[...] is already in the string table, but P[...]P is not. The compressor will parse out P[...], and find that P[...]P is not in the string table. It will output the code for P[...], and add P[...]P to the string table. Then it will get up to

P[...]P for the next string, and find that P[...]P is in the table, as the code just added. So it will output the code for P[...]P if it finds that P[...]PQ is not in the table. The decompressor is always "one step behind" the compressor. When the decompressor sees the code for P[...]P, it will not have added that code to it's string table yet because it needed the beginning character of P[...]P to add to the string for the last code, P[...], to form the code for P[...]P. However, when a decompressor finds a code that it doesn't know yet, it will always be the very next one to be added to the string table. So it can guess at what the string for the code should be, and, in fact, it will always be correct. If I am a decompressor, and I see code#124, and yet my string table has entries only up to code#123, I can figure out what code#124 must be, add it to my string table, and output the string. If code#123 generated the string, which I will refer to here as a prefix, [...], then code#124, in this special case, will be [...] plus the first character of [...]. So just add the first character of [...] to the end of itself. Not too bad. As an example (and a very common one) of this special case, let's assume we have a raster image in which the first three pixels have the same color value. That is, my charstream looks like: QQQ.... For the sake of argument, let's say we have 32 colors, and Q is the color#12. The compressor will generate the code sequence 12,32,.... (if you don't know why, take a minute to understand it.) Remember that #32 is not in the initial table, which goes from #0 to #31. The decompressor will see #12 and translate it just fine as color Q. Then it will see #32 and not yet know what that means. But if it thinks about it long enough, it can figure out that QQ should be entry#32 in the table and QQ should be the next string output. So the decompression pseudo-code goes something like:

```
 [1] Initialize string table;
[2] get first code: <code>;
[3] output the string for <code> to the charstream;
[4] <old> = <code>;
[5] <code> <- next code in codestream;
[6] does <code> exist in the string table?
 (yes: output the string for <code> to the charstream;
       [...] <- translation for <old>;
       K <- first character of translation for <code>;
       add [...]K to the string table;        <old> <- <code>;  )
 (no: [...] <- translation for <old>;
      K <- first character of [...];
      output [...]K to charstream and add it to string table;
      <old> <- <code>
 )
[7] go to [5];
```

Again, when you get to step [5] and there are no more codes, you're finished. Outputting of strings, and finding of initial characters in strings are efficiency problems all to themselves, but I'm not going to suggest ways to do them here. Half the fun of programming is figuring these things out!

Now for the GIF variations on the theme. In part of the header of a GIF file, there is a field, in the Raster Data stream, called "code size". This is a very misleading name for the field, but we have to live with it. What it is really is the "root size". The actual size, in bits, of the compression codes actually changes during compression/decompression, and I will refer to that size here as the "compression size". The initial table is just the codes for all the roots, as usual, but two special codes are added on top of those. Suppose you have a "code size", which is usually the number of bits per pixel in the image, of N. If the number of bits/pixel is one, then N must be 2: the roots take up slots #0 and #1 in the initial table, and the two special codes will take up slots #4 and #5. In any other case, N is the number of bits per pixel, and the roots take up slots #0 through #(2**N-1), and the special codes are (2**N) and (2**N + 1). The initial compression size will be N+1 bits per code. If you're encoding, you output the codes (N+1) bits at a time to start with, and if you're decoding, you grab (N+1) bits from the codestream at a time. As for the special codes: <CC> or the clear code, is (2**N), and <EOI>, or end-of-information, is (2**N + 1). <CC> tells the compressor to re-initialize the string table, and to reset the compression size to (N+1). <EOI> means there's no more in the codestream. If you're encoding or decoding, you should start adding things to the string table at <CC> + 2. If you're encoding, you should output <CC> as the very first code, and then whenever after that you reach code #4095 (hex FFF), because GIF does not allow compression sizes to be greater than 12 bits. If you're decoding, you should reinitialize your string table when you observe <CC>. The variable

compression sizes are really no big deal. If you're encoding, you start with a compression size of (N+1) bits, and, whenever you output the code (2**(compression size)-1), you bump the compression size up one bit. So the next code you output will be one bit longer. Remember that the largest compression size is 12 bits, corresponding to a code of 4095. If you get that far, you must output <CC> as the next code, and start over. If you're decoding, you must increase your compression size AS SOON AS YOU write entry #(2**(compression size) - 1) to the string table. The next code you READ will be one bit longer. Don't make the mistake of waiting until you need to add the code (2**compression size) to the table. You'll have already missed a bit from the last code. The packaging of codes into a bitsream for the raster data is also a potential stumbling block for the novice encoder or decoder. The lowest order bit in the code should coincide with the lowest available bit in the first available byte in the codestream. For example, if you're starting with 5-bit compression codes, and your first three codes are, say, <abcde>, <fghij>, <klmno>, where e, j, and o are bit#0, then your codestream will start off like:

```
byte#0: hijabcde
byte#1: .klmnofg
```

So the differences between straight LZW and GIF LZW are: two additional special codes and variable compression sizes. If you understand LZW, and you understand those variations, you understand it all!

Just as sort of a P.S., you may have noticed that a compressor has a little bit of flexibility at compression time. I specified a "greedy" approach to the compression, grabbing as many characters as possible before outputting codes. This is, in fact, the standard LZW way of doing things, and it will yield the best compression ratio. But there's no rule saying you can't stop anywhere along the line and just output the code for the current prefix, whether it's already in the table or not, and add that string plus the next character to the string table. There are various reasons for wanting to do this, especially if the strings get extremely long and make hashing difficult. If you need to, do it.

Hope this helps out.----steve blackstock

New Technical Notes

Macintosh

®

Developer Support

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.

MacDraw Format
Every couple of months someone requests this information on info-mac. Attached is the front end to a MacDraw-to-Imagen translator. It is in the C language header format. Microsoft graphics programs also output MacDraw format.

```
/*
 *      Description of MacDraw file
 *
 *      <MacDraw file> ::= HeadPacket <ObjectList> <End Object>
 *      <Object List> ::= <Object List> <Object> | <Object>
 *      <Object> ::= <Complex Object> | <Simple Object>
 *      <Complex Object> ::= <Nest Object> <Object List> <End Object>
 *      <Simple Object> ::= HeadWord <Body>
 *      <Object Body> ::= endObject | textObject | gridlineObject |
 *                        lineObject | rectObject | roundrectObject |
 *                        ovalObject | arcObject | freehandObject |
 *                        polyObject | nestObject
 *      <Nest Object> ::= HeadWord nestObject
 *      <End Object> ::= HeadWord endObject
 */

/* integer types */
typedef unsigned char   int8;
typedef short int       int16;
typedef long int        int32;

#define NOBJECTS        11

/* packet at head of MacDraw file */
struct  HeadPacket      {
        int16   unknown1[85];
        int16   PlotWidth;
        int16   PlotHeight;
        int16   PageWidth;
        int16   PageHeight;
        int16   unknown2[167];
        } HeadPacket;

/* word at beginning of each graphical object */
struct  HeadWord        {
        int8    ObjectType;
        int8    Lock;
        int16   unknown;
        } HeadWord;

/* ObjectType values */
#define endObject       0
#define textObject      1
#define gridlineObject  2
#define lineObject      3
#define rectObject      4
#define roundrectObject 5
#define ovalObject      6
#define arcObject       7
#define freehandObject  8
#define polyObject      9
#define nestObject      10
/* Object #11, Paint format bitmaps is not defined here */

/* Lock values */
#define unlocked        0
#define locked          1

/* end object delimiter */
struct  End     {
        int8    LineFat;
        int8    LinePat;
        int8    FillPat;
        int8    unknown;
        } End;

/* LineFat values */
#define NFat            6
```

```
#define invisibleLine   1
#define thinLine        2
#define mediumLine      3
#define thickLine       4
#define fatLine         5
#define defaultLine     2
/* fatness in rasters */
float FatTable[NFat] = {0.,0.,1.,2.,3.5,5.};

/* LinePat, FillPat values */
#define NPat               37
#define noPat               1
#define whitePat            2
#define blackPat            3
#define darkgrayPat         4
#define medgrayPat          5
#define lightgrayPat        6
#define coarsedotsPat       7
#define dotsPat             8
#define sparsedotsPat       9
#define topshinglePat      10
#define brickPat           11
#define slantbrickPat      12
#define leftdiagPat        13
#define thickleftdiagPat   14
#define dashleftdiagPat    15
#define narrowleftdiagPat  16
#define heavyleftdiagPat   17
#define dualdiagPat        18
#define horzdashpat        19
#define horzlinePat        20
#define circlePat          21
#define fourwayPat         22
#define smallhatchedPat    23
#define smalldiamondPat    24
#define rightdiagPat       25
#define thickrightdiagPat  26
#define dashrightdiagPat   27
#define narrowrightdiagPat 28
#define heavyrightdiagPat  29
#define trianglePat        30
#define vertdashpat        31
#define vertlinePat        32
#define rightshinglePat    33
#define heartPat           34
#define largehatchedPat    35
#define largediamondPat    36

/* pattern masks */
#define MPat                    8
unsigned char Pat[NPat][MPat] = {
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
0xBB,0xEE,0xBB,0xEE,0xBB,0xEE,0xBB,0xEE,
0x55,0xAA,0x55,0xAA,0x55,0xAA,0x55,0xAA,
0x88,0x22,0x88,0x22,0x88,0x22,0x88,0x22,
0x88,0x00,0x22,0x00,0x88,0x00,0x22,0x00,
0x80,0x00,0x08,0x00,0x80,0x00,0x08,0x00,
0x08,0x00,0x00,0x00,0x80,0x00,0x00,0x00,
0x80,0x80,0x41,0x3E,0x08,0x08,0x14,0xE3,
0x08,0x1C,0x22,0x41,0x80,0x01,0x02,0x04,
0xFF,0x80,0x80,0x80,0xFF,0x08,0x08,0x08,
0x01,0x80,0x40,0x20,0x10,0x08,0x04,0x02,
0x81,0xC0,0x60,0x30,0x18,0x0C,0x06,0x03,
0x11,0x88,0x44,0x00,0x11,0x88,0x44,0x00,
0x11,0x88,0x44,0x22,0x11,0x88,0x44,0x22,
0x33,0x99,0xCC,0x66,0x33,0x99,0xCC,0x66,
0x01,0x80,0x40,0x00,0x02,0x04,0x08,0x00,
0x66,0x00,0x00,0x00,0x99,0x00,0x00,0x00,
0xFF,0x00,0x00,0x00,0xFF,0x00,0x00,0x00,
0x50,0x20,0x20,0x20,0x50,0x88,0x27,0x88,
```

```
0x84,0x9F,0x80,0x80,0x04,0x04,0xE7,0x84,
0x01,0x01,0x01,0xFF,0x01,0x01,0x01,0xFF,
0x55,0x88,0x55,0x22,0x55,0x88,0x55,0x22,
0x80,0x01,0x02,0x04,0x08,0x10,0x20,0x40,
0xC0,0x81,0x03,0x06,0x0C,0x18,0x30,0x60,
0x88,0x11,0x22,0x00,0x88,0x11,0x22,0x00,
0x88,0x11,0x22,0x44,0x88,0x11,0x22,0x44,
0xCC,0x99,0x33,0x66,0xCC,0x99,0x33,0x66,
0x20,0x50,0x00,0x00,0x02,0x05,0x00,0x00,
0x08,0x08,0x08,0x08,0x08,0x08,0x08,0x08,
0x04,0x04,0x40,0x40,0x04,0x04,0x40,0x40,
0x03,0x84,0x48,0x30,0x0C,0x02,0x01,0x01,
0x0A,0x11,0xA0,0x40,0x00,0xB1,0x4A,0x4A,
0x40,0x40,0x40,0xFF,0x40,0x40,0x40,0x40,
0x41,0x22,0x14,0x08,0x14,0x22,0x41,0x80
};

/* text object */
struct  Text    {
        int8    LineFat;
        int8    LinePat;
        int8    FillPat;
        int8    unknown1;
        int16   BoxDx;
        int16   BoxDy;
        int8    Style;
        int8    Font;
        int8    Size;
        int8    LineSpace;
        int8    Justify;
        int8    Orient;
        int8    unknown2;
        int8    CharCount;
        int16   Top;
        int16   Left;
        int16   Bottom;
        int16   Right;
        /* plus CharCount bytes */
        } Text;
char    TextString[256];

/* Style values */
#define plainStyle      0
#define boldStyle       1
#define italicStyle     2
#define underlineStyle  4
#define outlineStyle    8
#define shadowStyle     16
#define defaultStyle    0

/* Font values */
#define ChicagoFont     1
#define GenevaFont      2
#define NewYorkFont     3
#define MonocoFont      4
#define VeniceFont      5
#define LondonFont      6
#define AthensFont      7
#define defaultFont     1

/* Size values */
#define size9           1
#define size10          2
#define size12          3
#define size14          4
#define size18          5
#define size24          6
#define size36          7
#define size48          8
#define NTextSize       9

/* LineSpace values */
#define singleSpace     1
```

```
#define halfSpace       2
#define doubleSpace     3
#define defaultSpace    1

/* Justify values */
#define leftJustify     1
#define centerJustify   2
#define rightJustify    3
#define defaultJustify  1

/* Orient values */
#define deg0Orient              0
#define deg90Orient             3
#define deg180Orient            2
#define deg270Orient            1
#define reflect0Orient          4
#define reflect90Orient         6
#define reflect180Orient        5
#define reflect270Orient        7

/* grid line object */
struct  GridLine        {
        int8    LineFat;
        int8    LinePat;
        int8    FillPat;
        int8    Arrow;
        int16   y1;
        int16   unknown1;
        int16   x1;
        int16   unknown2;
        int16   y2;
        int16   unknown3;
        int16   x2;
        int16   unknown4;
        } GridLine;

/* Arrow values */
#define noArrow         0
#define rightArrow      1
#define leftArrow       2
#define bothArrow       3
#define defaultArrow    0
/* arrow length in rasters */
float ArrowSize[NFat] = {0.,0.,5.,10.,17.,25.};
/* arrow angle in radians */
#define ARROW_ANGLE     .5

/* line object */
struct  Line    {
        int8    LineFat;
        int8    LinePat;
        int8    FillPat;
        int8    Arrow;
        int16   y1;
        int16   unknown1;
        int16   x1;
        int16   unknown2;
        int16   y2;
        int16   unknown3;
        int16   x2;
        int16   unknown4;
        } Line;

/* rectangle object */
struct  Rect    {
        int8    LineFat;
        int8    LinePat;
        int8    FillPat;
        int8    Corner;
        int16   Top;
        int16   unknown1;
        int16   Left;
        int16   unknown2;
```

```
        int16   Bottom;
        int16   unknown3;
        int16   Right;
        int16   unknown4;
        } Rect;

/* Corner values */
#define NCorner         6
#define zeroCorner      0
#define one8Corner      1
#define three16Corner   2
#define one4Corner      3
#define five16Corner    4
#define three8Corner    5
/* radii in inches */
float RadiusTable[NCorner] = {0.,.125,.1875,.25,.3125,.375};

/* rounded rectangle object */
struct  RoundRect       {
        int8    LineFat;
        int8    LinePat;
        int8    FillPat;
        int8    Corner;
        int16   Top;
        int16   unknown1;
        int16   Left;
        int16   unknown2;
        int16   Bottom;
        int16   unknown3;
        int16   Right;
        int16   unknown4;
        } RoundRect;

/* oval object */
struct  Oval    {
        int8    LineFat;
        int8    LinePat;
        int8    FillPat;
        int8    unknown;
        int16   Top;
        int16   unknown1;
        int16   Left;
        int16   unknown2;
        int16   Bottom;
        int16   unknown3;
        int16   Right;
        int16   unknown4;
        } Oval;

/* arc object */
struct  Arc     {
        int8    LineFat;
        int8    LinePat;
        int8    FillPat;
        int8    unknown;
        int16   Top;
        int16   unknown1;
        int16   Left;
        int16   unknown2;
        int16   Bottom;
        int16   unknown3;
        int16   Right;
        int16   unknown4;
        int16   StartAngle;
        int16   NDegree;
        } Arc;

/* point objects */
#define NPoint          256
struct  Point   {
        int16   y;
        int16   unknown1;
        int16   x;
```

```
        int16   unknown2;
        } Point[NPoint];

struct Delta    {
        char    dx;
        char    dy;
        } Delta[NPoint];

/* freehand line object */
struct  FreeHand        {
        int8    LineFat;
        int8    LinePat;
        int8    FillPat;
        int8    unknown1;
        int16   unknown2;
        int16   Bytes;
        int16   PointCount;
        int16   Top;
        int16   unknown3;
        int16   Left;
        int16   unknown4;
        int16   Bottom;
        int16   unknown5;
        int16   Right;
        int16   unknown6;
        int16   unknown7;
        int16   y1;
        int16   unknown8;
        int16   x1;
        int16   unknown9;
        /* plus Bytes-28 bytes or Bytes/2-14 dx,dy pairs */
        } FreeHand;

/* polygon object */
struct  Poly    {
        int8    LineFat;
        int8    LinePat;
        int8    FillPat;
        int8    unknown1;
        int16   unknown2;
        int16   Bytes;
        int16   PointCount;
        int16   unknown3;
        int16   unknown4;
        int16   Top;
        int16   unknown5;
        int16   Left;
        int16   unknown6;
        int16   Bottom;
        int16   unknown7;
        int16   Right;
        /* plus Bytes-20 or PointCount*4 bytes, PointCount x,y pairs */
        } Poly;

/* nest object delimiter */
struct  Nest    {
        int8    LineFat;
        int8    LinePat;
        int8    FillPat;
        int8    unknown1;
        int16   unknown2;
        int16   ObjectCount;
        int16   unknown3;
        int16   Bytes;
        int16   Top;
        int16   unknown4;
        int16   Left;
        int16   unknown5;
        int16   Bottom;
        int16   unknown6;
        int16   Right;
        int16   unknown7[5];
        } Nest;
```

```
/*
 *      count Object lengths
 *
 *      #include "MacDraw.h"
 *      main(){
 *              printf ("HeadPacket=%d\n",sizeof(HeadPacket));
 *              printf ("HeadWord=%d\n",sizeof(HeadWord));
 *              printf ("End=%d\n",sizeof(End));
 *              printf ("Text=%d\n",sizeof(Text));
 *              printf ("GridLine=%d\n",sizeof(GridLine));
 *              printf ("Line=%d\n",sizeof(Line));
 *              printf ("Rect=%d\n",sizeof(Rect));
 *              printf ("RoundRect=%d\n",sizeof(RoundRect));
 *              printf ("Oval=%d\n",sizeof(Oval));
 *              printf ("Arc=%d\n",sizeof(Arc));
 *              printf ("FreeHand=%d\n",sizeof(FreeHand));
 *              printf ("Poly=%d\n",sizeof(Poly));
 *              printf ("Nest=%d\n",sizeof(Nest));
 *              };
 */
```

Standard MIDI File Format

Dustin Caldwell

The standard MIDI file format is a very strange beast. When viewed as a whole, it can be quite overwhelming. Of course, no matter how you look at it, describing a piece of music in enough detail to be able to reproduce it accurately is no small task. So, while complicated, the structure of the midi file format is fairly intuitive when understood.

I must insert a disclaimer here that I am by no means an expert with midi nor midi files. I recently obtained a Gravis UltraSound board for my PC, and upon hearing a few midi files (.MID) thought, "Gee, I'd like to be able to make my own .MID files." Well, many aggravating hours later, I discovered that this was no trivial task. But, I couldn't let a stupid file format stop me. (besides, I once told my wife that computers aren't really that hard to use, and I'd hate to be a hypocrite) So if any errors are found in this information, please let me know and I will fix it. Also, this document's scope does not extend to EVERY type of midi command and EVERY possible file configuration. It is a basic guide that should enable the reader (with a moderate investment in time) to generate a quality midi file.

1. Overview

A midi (.MID) file contains basically 2 things, Header chunks and Track chunks. Section 2 explains the header chunks, and Section 3 explains the track chunks. A midi file contains ONE header chunk describing the file format, etc., and any number of track chunks. A track may be thought of in the same way as a track on a multi-track tape deck. You may assign one track to each voice, each staff, each instrument or whatever you want.

2. Header Chunk

The header chunk appears at the beginning of the file, and describes the file in three ways. The header chunk always looks like:

```
4D 54 68 64 00 00 00 06 ff ff nn nn dd dd
```

The ascii equivalent of the first 4 bytes is MThd. After MThd comes the 4-byte size of the header. This will always be 00 00 00 06, because the actual header information will always be 6 bytes.

ff ff is the file format. There are 3 formats:

0 - single-track
1 - multiple tracks, synchronous
2 - multiple tracks, asynchronous

Single track is fairly self-explanatory - one track only. Synchronous multiple tracks means that the tracks will all be vertically synchronous, or in other words, they all start at the same time, and so can represent different parts in one song. Asynchronous multiple tracks do not necessarily start at the same time, and can be completely asynchronous.

nn nn is the number of tracks in the midi file.

dd dd is the number of delta-time ticks per quarter note. (More about this later)

3. Track Chunks

The remainder of the file after the header chunk consists of track chunks. Each track has one header and may contain as many midi commands as you like. The header for a track is very similar to the one for the file:

```
4D 54 72 6B xx xx xx xx
```

As with the header, the first 4 bytes has an ascii equivalent. This one is MTrk. The 4 bytes after MTrk give the length of the track (not including the track header) in bytes.

Following the header are midi events. These events are identical to the actual data sent and received by MIDI ports on a synth with one addition. A midi event is preceded by a delta-time. A delta time is the number of ticks

after which the midi event is to be executed. The number of ticks per quarter note was defined previously in the file header chunk. This delta-time is a variable-length encoded value. This format, while confusing, allows large numbers to use as many bytes as they need, without requiring small numbers to waste bytes by filling with zeros. The number is converted into 7-bit bytes, and the most-significant bit of each byte is 1 except for the last byte of the number, which has a msb of 0. This allows the number to be read one byte at a time, and when you see a msb of 0, you know that it was the last (least significant) byte of the number. According to the MIDI spec, the entire delta-time should be at most 4 bytes long.

Following the delta-time is a midi event. Each midi event (except a running midi event) has a command byte which will always have a msb of 1 (the value will be >= 128). A list of most of these commands is in appendix A. Each command has different parameters and lengths, but the data that follows the command will have a msb of 0 (less than 128). The exception to this is a meta-event, which may contain data with a msb of 1. However, meta-events require a length parameter which alleviates confusion.

One subtlety which can cause confusion is running mode. This is where the actual midi command is omitted, and the last midi command issued is assumed. This means that the midi event will consist of a delta-time and the parameters that would go to the command if it were included.

4. Conclusion

If this explanation has only served to confuse the issue more, the appendices contain examples which may help clarify the issue. Also, 2 utilities and a graphic file should have been included with this document:

DEC.EXE - This utility converts a binary file (like .MID) to a tab-delimited text file containing the decimal equivalents of each byte.

REC.EXE - This utility converts a tab-delimited text file of decimal values into a binary file in which each byte corresponds to one of the decimal values.

MIDINOTE.PS - This is the postscript form of a page showing note numbers with a keyboard and with the standard grand staff.

Appendix A

1. MIDI Event Commands

Each command byte has 2 parts. The left nybble (4 bits) contains the actual command, and the right nybble contains the midi channel number on which the command will be executed. There are 16 midi channels, and 8 midi commands (the command nybble must have a msb of 1).
In the following table, x indicates the midi channel number. Note that all data bytes will be <128 (msb set to 0).

Hex	Binary	Data	Description
8x	1000xxxx	nn vv	Note off (key is released) nn=note number vv=velocity
9x	1001xxxx	nn vv	Note on (key is pressed) nn=note number vv=velocity
Ax	1010xxxx	nn vv	Key after-touch nn=note number vv=velocity
Bx	1011xxxx	cc vv	Control Change cc=controller number vv=new value
Cx	1100xxxx	pp	Program (patch) change pp=new program number
Dx	1101xxxx	cc	Channel after-touch cc=channel number
Ex	1110xxxx	bb tt	Pitch wheel change (2000H is normal or no

			change) bb=bottom (least sig) 7 bits of value tt=top (most sig) 7 bits of value

The following table lists meta-events which have no midi channel number. They are of the format:

FF xx nn dd

All meta-events start with FF followed by the command (xx), the length, or number of bytes that will contain data (nn), and the actual data (dd).

Hex	Binary	Data	Description
00	00000000	nn ssss	Sets the track's sequence number. nn=02 (length of 2-byte sequence number) ssss=sequence number
01	00000001	nn tt ..	Text event- any text you want. nn=length in bytes of text tt=text characters
02	00000010	nn tt ..	Same as text event, but used for copyright info. nn tt=same as text event
03	00000011	nn tt ..	Sequence or Track name nn tt=same as text event
04	00000100	nn tt ..	Track instrument name nn tt=same as text event
05	00000101	nn tt ..	Lyric nn tt=same as text event
06	00000110	nn tt ..	Marker nn tt=same as text event
07	00000111	nn tt ..	Cue point nn tt=same as text event
2F	00101111	00	This event must come at the end of each track
51	01010001	03 tttttt	Set tempo tttttt=microseconds/quarter note
58	01011000	04 nn dd ccbb	Time Signature nn=numerator of time sig. dd=denominator of time sig. 2=quarter 3=eighth, etc. cc=number of ticks in metronome click bb=number of 32nd notes to the quarter note
59	01011001	02 sf mi	Key signature sf=sharps/flats (-7=7 flats, 0=key of C, 7=7 sharps) mi=major/minor (0=major, 1=minor)
7F	01111111	xx dd ..	Sequencer specific information xx=number of bytes to be sent dd=data

The following table lists system messages which control the entire system. These have no midi channel number. (these will generally only apply to controlling a midi keyboard, etc.)

Hex	Binary	Data	Description
F8	11111000		Timing clock used when synchronization is required.
FA	11111010		Start current sequence

```
FB       11111011                  Continue a stopped sequence where left
                                   off

FC       11111100                  Stop a sequence
```

The following table lists the numbers corresponding to notes for use in note on and note off commands.

Octave #	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				

BIBLIOGRAPHY

"MIDI Systems and Control" Francis Rumsey 1990 Focal Press

"MIDI and Sound Book for the Atari ST" Bernd Enders and Wolfgang Klemme 1989 M&T Publishing, Inc.

MIDI file specs and general MIDI specs were also obtained by sending e-mail to LISTSERV@AUVM.AMERICAN.EDU with the phrase GET MIDISPEC PACKAGE in the message.

------------------------------ DEC.CPP -----------------------------------

```
/*  file  dec.cpp

by  Dustin Caldwell    (dustin@gse.utah.edu)

*/


#include <dos.h>
#include <stdio.h>
#include <stdlib.h>

void helpdoc();

main()
{
        FILE *fp;

        unsigned char ch, c;

        if((fp=fopen(_argv[1], "rb"))==NULL)            /* open file to read */
        {
                printf("cannot open file %s\n",_argv[1]);
                helpdoc();
                exit(-1);
        }

        c=0;
        ch=fgetc(fp);

        while(!feof(fp))                        /* loop for whole file */
        {
                printf("%u\t", ch);             /* print every byte's decimal equiv. */
```

```
		c++;
		if(c>8)					/* print 8 numbers to a line */
		{
			c=0;
			printf("\n");
		}

		ch=fgetc(fp);
	}

	fclose(fp);			/* close up */
}

void helpdoc()			/* print help message */
{
	printf("\n   Binary File Decoder\n\n");

	printf("\n Syntax:  dec binary_file_name\n\n");

	printf("by Dustin Caldwell  (dustin@gse.utah.edu)\n\n");
	printf("This is a filter program that reads a binary file\n");
	printf("and prints the decimal equivalent of each byte\n");
	printf("tab-separated. This is mostly useful when piped \n");
	printf("into another file to be edited manually.  eg:\n\n");
	printf("c:\>dec sonata3.mid > son3.txt\n\n");
	printf("This will create a file called son3.txt which can\n");
	printf("be edited with any ascii editor. \n\n");
	printf("(rec.exe may also be useful, as it reencodes the \n");
	printf("ascii text file).\n\n");
	printf("Have Fun!!\n");
}

---------------------------- REC.CPP ----------------------------------

/*  File  rec.cpp
	by Dustin Caldwell   (dustin@gse.utah.edu)
*/

#include <dos.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

void helpdoc();

main()
{
	FILE *rfp, *wfp;

	unsigned char ch, c;
	char s[20];

	if((rfp=fopen(_argv[1], "r"))==NULL)			/* open the read file */
	{
		printf("cannot open file %s \n",_argv[1]);
		helpdoc();
		exit(-1);
	}

	if((wfp=fopen(_argv[2], "wb"))==NULL)			/* open the write file */
	{
		printf("cannot open file %s \n",_argv[1]);
		helpdoc();
		exit(-1);
	}

	c=0;

	ch=fgetc(rfp);

	while(!feof(rfp))				/* loop for whole file */
	{
```

```
		if(isalnum(ch))			/* only 'see' valid ascii chars */
		{
			c=0;
			while(isdigit(ch))	/* only use decimal digits (0-9) */
			{
				s[c]=ch;	/* build a string containing the number */
				c++;
				ch=fgetc(rfp);
			}
			s[c]=NULL;			/* must have NULL terminator */

			fputc(atoi(s), wfp);/* write the binary equivalent to file */

		}

		ch=fgetc(rfp);			/* loop until next number starts */


	}

	fclose(rfp);			/* close up */
	fclose(wfp);
}


void helpdoc()		/* print help message */
{
	printf("\n   Text File Encoder\n\n");

	printf("\n Syntax:  rec text_file_name binary_file_name\n\n");

	printf("by Dustin Caldwell  (dustin@gse.utah.edu)\n\n");
	printf("This is a program that reads an ascii tab-\n");
	printf("delimited file and builds a binary file where\n");
	printf("each byte of the binary file is one of the decimal\n");
	printf("digits in the text file.\n");
	printf(" eg:\n\n");
	printf("c:\>rec son3.txt son3.mid\n\n");
	printf("(This will create a file called son3.mid which is\n");
	printf("a valid binary file)\n\n");
	printf("(dec.exe may also be useful, as it decodes binary files)\n\n");
	printf("Have Fun!!\n");
}
```

From: Wilson Woo <wilson00@HK.Super.NET>
To: submit@wotsit.demon.co.uk
Subject: MPEG Video

THIS TEXT CONTAINS ONLY MPEG VIDEO HEADER INFO - BY WILSON WOO
It's only what I know. Please feel free to update it.

Below is information got from someone.

/***/

Sequence Header

This contains information related to one or more "group-of-pictures"

Byte#	Data	Details
1-4	Sequence header code	In Hex 000001B3
12 bits	Horizontal size	In pixels
12 bits	Vertical size	In pixels
4 bits	Pel aspect ratio	See below
18 bits	Picture rate	See below
1 bit	Marker bit	Always 1
10 bits	VBV buffer size	Minimum buffer needed to decode this sequence of pictures; in 16KB units
1 bit	Constrained parameter flag	
1 bit	Load intra quantizer matrix	0: false; 1: true (matrix follows)
64 bytes	Intra quantizer matrix	Optional
1 bit	Load nonintra quantizer matrix	0: false; 1: true (matrix follows)
64 bytes	Nonintra quantizer matrix	Optional
-	Squence extension Data	Optional
-	User data	Optional application-dependent data

Aspect raios are defined by a code which represents the height and width of the Video image.
Picture rates are also defined by a code that represents the number of pictures that may be displayed each second.

Each group of pictures has a header that contains one "I picture" and zero or more B and P pictures. The header is concerned with the time synchronisation for the first picture in this group, and the closeness of the previous group to this one.

/***/

For picture rate:
1 = 23.976 frames/sec
2 = 24
3 = 25
4 = 29.97
5 = 30
6 = 50
7 = 59.94
8 = 60

Here gives an example. Below is Hex dump of first 256 bytes of the first Video frame of TEST.MPG from XingMPEG.

```
00 00 01 B3 16 00 F0 C4 02 A3 20 A5 10 12 12 14
14 14 16 16 16 16 18 18 19 18 18 1A 1B 1B 1B 1B
1A 1C 1D 1E 1E 1E 1D 1C 1E 1F 20 21 21 20 1F 1E
21 23 23 24 23 23 21 25 26 27 27 26 25 29 2A 2A
2A 29 2D 2D 2D 2D 30 31 30 34 34 38 16 00 F0 C4
00 00 01 B8 00 08 00 00 00 00 01 00 00 0A 72 00
00 00 01 01 13 F9 50 02 BC B2 B8 BE 68 8B A4 9F
```

```
C5 B5 CA 00 56 76 39 65 F2 30 8B A6 9D 50 69 E7
DA FE 13 CF B7 FF 8F F4 CE 7B FA 0E F0 66 AE 1C
5D E7 00 C8 0A 92 B9 29 3C 21 23 F1 D6 40 13 06
F0 10 10 C6 27 80 A0 34 E1 C8 E4 0F 74 91 DA C4
03 A0 DC 03 12 60 18 49 27 1D D4 BC 67 0E 54 8C
96 FC 5D C0 06 E0 1A 72 11 7C 9A 8D C9 45 89 6D
CD C4 0B 63 DC 90 18 24 00 EC 84 90 18 10 C9 3B
1E A7 60 3C 9D 74 80 76 05 0B 02 81 A9 29 39 68
53 8F 59 F1 BF 93 FB A0 04 01 BC B0 CE 18 E1 25

Sequence header  = (Hex) 00 00 01 B3
Horizontal size  = 0x160 = 352
Vertical size    = 0x0F0 = 240
Pel aspect ratio = [I don't know]
Picture rate     = 4 = 29.97 frames/sec
Marker bit       = 1
```

MPEG-2 Technical (and sometimes political) Frequently Asked Questions
(FAQ) list.
Copyright 1994 by Chad Fogg (cfogg@netcom.com)
Draft 3.3 (May 10, 1994)

1. MPEG is a DCT based scheme, right?
2. What does the MPEG video syntax feature that codes video efficiently?
3. What does the syntax provide for error robustness?
4. What is the significance of each layer in MPEG video ?
5. How does the syntax facilitate parallelism?
6. I hear the encoder is not part of the standard?
7. Are some encoders better than others?
8. Can MPEG-1 encode higher sample rates than 352 x 240 x 30 Hz ?
9. What are Constrained Parameters Bitstreams (CPB) for video?
10. Why is Constrained Parameters so important?
11. Who uses constrained parameters bitstreams?
12. Are there ways of circumventing constrained parameters bitstreams for SIF
class applications and decoders ?
13. Are there any other conformance points like CPB for MPEG-1?
14. What frame rates are permitted in MPEG?
15. Special prediction switches for MPEG-2
16. What is MPEG-2 Video Main Profile and Main Level?
17. Does anybody actually use the scalability modes?
18. What's the difference between Field and Frame pictures?
19. What do B-pictures buy you?
20. Why do some people hate B-frames?
21. Why was the 16x16 area chosen?
22. Why was the 8x8 DCT size chosen?
23. What is motion compensated prediction, and why is it a pain?
24. What are the various prediction modes in MPEG-2?
24.1 Frame:
24.2 Field predictions in frame-coded pictures:
24.3 Field predictions in field-coded pictures:
24.4 16x8 predictions in field-coded pictures:
24.5 Dual Prime prediction in frame and field-coded pictures
24.6 Field and frame organized macroblocks:
25. How do you tell a MPEG-1 bitstream from a MPEG-2 bitstream?
26. What is the reasoning behind MPEG syntax symbols?
27. Why bother to research compressed video when there is a standard?
28. Where can I get a copy of the latest MPEG-2 draft?
29. What are the latest working drafts of MPEG-2 ?
30. What is the latest version of the MPEG-1 documents?
31. What is the evolution of ISO standard documents?
32. Where is a good introductory paper to MPEG?
33. What are some journals on related MPEG topics ?
34. Is there a book on MPEG video?
35. Is it MPEG-2 (Arabic numbers) or MPEG-II (roman)?
36. What happened to MPEG-3?
37. What is MPEG-4?
38. What are the scaleable modes of MPEG-2?
39. Why MPEG-2? Wasn't MPEG-1 enough?
40. What did MPEG-2 add to MPEG-1 in terms of syntax/algorithms ?
41. How do MPEG and JPEG differ?
42. How do MPEG and H.261 differ?
43. Is H.261 the de facto teleconferencing standard?
44. What is the TM rate control and adaptive quantization technique ?
45. How does the TM work?
46. What is a good motion estimation method, then?
47. Is exhaustive search "optimal" ?
48. What are some advanced encoding methods?
49. Is so-and-so really MPEG compliant ?
50. What are the tell-tale MPEG artifacts?
51. Where are the weak points of MPEG video ?
52. What are some myths about MPEG?
53. What is the color space of MPEG?
54. Don't you mean 4:1:1 ?
55. Why did MPEG choose 4:2:0 ? Isn't 4:2:2 the standard for TV?
56. What is the precision of MPEG samples?
57. What is all the fuss with cositing of chroma components?
58. How would you explain MPEG to the data compression expert?
59. How does MPEG video really compare to TV, VHS, laserdisc ?
60. What are the typical MPEG-2 bitrates and picture quality?
61. At what bitrates is MPEG-2 video optimal?

62. Why does film perform so well with MPEG ?
63. What is the best compression ratio for MPEG ?
64. Can MPEG be used to code still frames?
65. Is there an MPEG file format?
66. What are some pre-processing enhancements ?
67. Why use these "advanced" pre-filtering techniques?
68. What about post-processing enhancements?
69. Can motion vectors be used to measure object velocity?
70. How do you code interlaced video with MPEG-1 syntax?
71. Is MPEG patented?
72. How many cable box alliances are there?
73. Will there be an MPEG video tape format?
74. Where will be see MPEG in everyday life?
75. What is the best compression ratio for MPEG ?
76. Is there a MPEG CD-ROM format?

1. MPEG is a DCT based scheme, right?

The DCT and Huffman algorithms receive the most press coverage (e.g. MPEG is a DCT based scheme with Huffman coding), but are in fact fairly insignificant. The variety of coding modes signaled to the decoder as context-dependent side information are chiefly responsible for the efficiency of the MPEG syntax.

2. What does the MPEG video syntax feature that codes video efficiently?

A. Here are some of the statistical conditions and their syntax counterparts.

Occlusion: forward, backwards, or bi-directional temporal prediction in B pictures.

Smooth optical flow fields: variable length coding of 1-D prediction errors for motion vectors.

Spatial correlation beyond 8x8 sample block boundaries: 1-D prediction of DC coefficients in consecutive group intra-coded macroblocks.

High temporal correlation: variable on/off coding of prediction error at the macroblock (no-coding) or individual block (coded block pattern) level.

Temporal de-correlation: forward, backwards, or bidirectional prediction.

Content dependent quality: locally adaptive quantization

Temporal prediction accuracy: "half-pel" sample accuracy.

High locally correlated signal refresh pictures (I picture) and prediction errors: DCT

Subjective coding: location-dependent quantization of DCT coefficients.

3. What does the syntax provide for error robustness?

1. Byte-aligned start codes in the coded bitstream.
2. End of block codes in coded blocks.
3. Slices.
4. slice_vertical_position embedded as sub-field within slice start codes.
5. slices commencing at regular locations in picture (MPEG-2)

4. What is the significance of each layer in MPEG video ?

Sequence:
Set of pictures sharing same sampling dimensions, bit rate, chromaticy (MPEG-1), quantization matrices (MPEG-1 only).

Group of Pictures:
Random access point giving SMPTE time code within sequence.
Guaranteed to start with an I picture.

Picture:
Samples of a common plane -- "captured" from the same time instant.

Slice:

Error resynchronization unit of macroblocks.
At the commencement of a slice, all inter-macroblock coding dependencies are reset. Likewise, all macroblocks within a common slice can be dependently coded.

Macroblock:

Least common multiple of Y, Cb, Cr 8x8 blocks in 4:2:0 sampling structure.
For MPEG-1, the smallest granularity of temporal prediction.

Block:

Smallest granularity of spatial decorrelation.

5. How does the syntax facilitate parallelism?

A. For MPEG-1, slices may consist of an arbitrary number of macroblocks. The coded bitstream must first be mapped into fixed-length elements before true parallelism in a decoder application can be exploited. Further, since macroblocks have coding dependencies on previous macroblocks within the same slice, the data hierarchy must be pre-processed down to the layer of DC DCT coefficients. After this, blocks may be independently inverse transformed and quantized, temporally predicted, and reconstructed to buffer memory. Parallelism is usually more of a concern for encoders. Macroblock motion estimation and some rate control stages can be processed independently. An encoder also has the freedom to choose the slice structure.

6. I hear the encoder is not part of the standard?

A. The encoder rests just outside the normative scope of the standard, as long as the bitstreams it produces are compliant. The decoder, however, is almost deterministic: a given bitstream should reconstruct to a unique set of pictures. Statistically speaking, an occasional error of a Least Significant Bit is permitted as a result of the fact that the IDCT function is the only non-normative stage in the decoder (the designer is free to choose among many DCT algorithms and implementations). The IEEE 1180 test referenced in Annex A of the MPEG-1 and MPEG-2 specifications spells out the statistical mismatch tolerance between the Reference IDCT, which uses 64-bit floating point accuracy, and the Test IDCT.

7. Are some encoders better than others?

A. Yes. For example, the range over which a compensated prediction macroblock is searched for has a great influence over final picture quality. At a certain point a very large range can actually become detrimental (it may encourage large differential motion vectors). Practical ranges are usually between +/- 15 and +/- 32. As the range doubles, for instance, the search area quadruples.

8. Can MPEG-1 encode higher sample rates than 352 x 240 x 30 Hz ?

A. Yes. The MPEG-1 syntax permits sampling dimensions as high as 4095 x 4095 x 60 frames per second. The MPEG most people think of as "MPEG-1" is actually a kind of subset known as Constrained Parameters bitstream (CPB).

9. What are Constrained Parameters Bitstreams (CPB) for video?

A. MPEG-1 CPB are a limited set of sampling and bitrate parameters designed to normalize decoder computational complexity, buffer size, and memory bandwidth while still addressing the widest possible range of applications. The parameter limits were intentionally designed so that a decoder implementation would need only 4 Megabits of DRAM.

Parameter	Limit
pixels/line	704
lines/picture	480 or 576
pixels*lines	352*240 or 352*288
picture rate	30 Hz

bit rate   1.862million bits/sec
buffer size   40 Kilobytes (327,680 bits)

The sampling limits of CPB are bounded at the ever popular SIF rate: 396 macroblocks (101,376 pixels) per picture if the picture rate is less than or equal to 25 Hz, and 330 macroblocks (84,480 pixels) per picture if the picture rate is 30 Hz. The MPEG nomenclature loosely defines a "pixel" or "pel" as a unit vector containing a complete luminance sample and one fractional (0.25 in 4:2:0 format) sample from each of the two chrominance (Cb and Cr) channels. Thus, the corresponding bandwidth figure can be computed as:

352 samples/line x 240 lines/picture x 30 pictures/sec x 1.5 samples/pixel

or 3.8 Ms/s (million samples/sec) including chroma, but not including blanking intervals. Since most decoders are capable of sustaining VLC decoding at a faster rate than 1.8 Mbit/sec, the coded video bitrate has become the most often waived parameter of CPB. An encoder which intelligently employs the syntax tools should achieve SIF quality saturation at about 2 Mbit/sec, whereas an encoder producing streams containing only I (Intra) pictures might require as much as 4 Mbit/sec to achieve the same video quality.

10. Why is Constrained Parameters so important?

A. It is an optimum point that allows (just barely) cost effective VLSI implementations in 1992 technology (0.8 microns). It also implies a nominal guarantee of interoperability for decoders and encoders. Since CPB is a canonical conformance point, MPEG devices which are not capable of meeting SIF rates are usually not considered to be true MPEG.

11. Who uses constrained parameters bitstreams?

A. Applications which are focused on CPB are Compact Disc (White Book or CD-I) and computer video applications. Set-top TV decoders fall into a higher sampling rate category known as CCIR 601 or Broadcast rate.

12. Are there ways of circumventing constrained parameters bitstreams for SIF class applications and decoders ?

A. Yes, some. Remember that CPB limits pictures by macroblock count. 416 x 240 x 24 Hz sampling rates are still within the constraints, but this would only be of benefit in NTSC (240 lines/field) displays. Deviating from 352 samples/line could throw off many decoder implementations which possess limited horizontal sample rate conversion abilities. Some decoders do in fact include a few rate conversion modes, with a filter usually implemented via binary taps (shifts and adds). Likewise, the target sample rates are usually limited or ratios (e.g. 640, 540, 480 pixels/line, etc.). Future MPEG decoders will likely include on-chip arbitrary sample rate converters, perhaps capable of operating in the vertical direction (although there is little need of this in applications using standard TV monitors, with the possible exception of windowing in cable box graphical user interfaces).

13. Are there any other conformance points like CPB for MPEG-1?
A. Undocumented ones, yes. A second generation of decoder chips emerged on the market about 1 year after the first wave of SIF-class decoders. Both LSI Logic and SGS-Thomson introduced CCIR 601 class MPEG-1 decodersto fill in the gap between canonical MPEG-1 and the emergence of MPEG-2. Under non-disclosure agreement, C-Cube had the CL-950.

14. What frame rates are permitted in MPEG?
A. A limited set is available for the choosing in MPEG-1, although "tricks" could be played with Systems-layer Time Stamps to convey non-standard rates. The set is: 23.976 Hz (3-2 pulldown NTSC), 24 Hz (Film), 25 Hz (PAL/SECAM or 625/60 video), 29.97 (NTSC), 30 Hz (drop-frame NTSC or component 525/60), 50 Hz (double-rate PAL), 59.97 Hz (double rate NTSC), and 60 Hz (double-rate drop-frame NTSC/component 525/60 video).

15. Special prediction switches for MPEG-2

MPEG-2 sequence

```
              /         \
  progressive           interlaced sequence
  sequence             /                     \
                Field picture            Frame picture
                                        /            \
                            Frame or field pred.     Frame MB prediction only
                              /         \
                        Field dct      Frame dct
```

16. What is MPEG-2 Video Main Profile and Main Level?

A. MPEG-2 Video Main Profile and Main Level is analogous to MPEG-1's CPB,with sampling limits at CCIR 601 parameters (720 x 480 x 30 Hz). Profiles limit syntax (i.e. algorithms), whereas Levels place limits on coding parameters (sample rates, frame dimensions, coded bitrates, etc.). Together, Video Main Profile and Main Level (abbreviated as MP@ML) normalize complexity within feasible limits of 1994 VLSI technology (0.5 micron), yet still meet the needs of the majority of application users.MP@ML is the conformance point for most cable and satellite systems.

Profiles
======
Simple: I and P pictures only. 4:2:0 sampling ratio. 8,9, or 10 bits DC precision.
Main: I, P, and B pictures. Dual Prime with no B-pictures only. 4:2:0 sampling ratio. 8, 9, or 10 bits sample precision.
SNR profile:
Spatial profile:
High: 8,9,10, or 11 bits sample precision. 4:2:2 and 4:4:4 sampling ratio.

Level
====
Simple: SIF video rate (3.041280 Mhz), 4 Mbit/sec, 0.489472 Mbit VBV buffer, 64 vertical in frame, 32Vertical in field, 1:7 fcode hor.

Main: CCIR 601 video rate (10.368 Mhz), 15 Mbit/sec, 1.835008 Mbit VBV buffer, 128 V in frame, 64 V in field, 1:8 f_code Hor.

High 1440: 1440 x 1152 x 30 Hz (47.0016 Mhz), 60 Mbit/sec. 7.340032 Mbit VBV buffer, 128 V in Fe, 1:9 fcode H.

High: 1920 x 1152 x 30 Hz (62.6688 Mhz), 80 Mbit/sec. 9.787392 Mbit VBV buffer.
1:9 fcode H

17. Does anybody actually use the scalability modes?

A. At this time, scalability has found itself a limited number of applications, although research is definitely underway for its use in HDTV. Experiments have been demonstrated in Europe where, for example, PAL-rate video (720 x 576 x 25 fps) is embedded in the same stream as HDTV rate video (1440 x 1152 x 25 fps). The Nov. 1992 VADIS experiment divided the base layer (PAL) and enhancement into 4 and 16 Mbit/sec channels, respectively. The U.S. Grand Alliance favors HDTV simulcasting (separate NTSC analog and digital HDTV broadcasts). Temporal scalability is the pet scalability mode as the possible future solution for coding 60 Hz progressive sequences while maintaining backwards compatibility with early-wave equipment (e.g. 1920 x 1080 x 30 Hz displays) . To elaborate, the first wave receivers of the late 1990's would be limited to 62at 0 Hz interlaced/30 Hz progressive HDTV decoders. Essentially, 60 interlaced fields would be coded in a, for example, 16 Mbit/sec stream in 1996, and when VLSI processes shift another thousand or so angstroms down the wavelength scale, an 8 Mbit/sec enhancement layer containing the coded "high pass" between 60 Hz progressive and 60 Hz interlaced would be simulcasted or multiplexed. Several corporate mouths have been known to water at the mention of charging the quality conscious subscriber an extra fee for the enhancement layer.

18. What's the difference between Field and Frame pictures?
A. A frame-coded picture consists of samples from both even and odd fields.
A
frame picture is coded in progressive order (an even line, then an odd line,

etc.) and in the case of MPEG-2, may optionally switch between field and frame order on a macroblock basis. The Display Process, which is *almost* completely outside the scope of the MPEG specification, can chose to re-interlace the picture by displaying the odd and even lines at different times (16 milliseconds apart for 60 Hz displays). In fact, most pictures, regardless of whether they were coded as a Field or Frame, end up being displayed interlaced due to the fact that most TV sets are interlaced.

19. What do B-pictures buy you?

A. Since bi-directional macroblock predictions are an average of two macroblock areas, noise is reduced at low bit rates (like a 3-D filter, if you will). At nominal MPEG-1 video (352 x 240 x 30, 1.15 Mbit/sec) rates, it is said that B-frames improves SNR by as much as 2 dB. (0.5 dB gain is usually considered worth-while in MPEG). However, at higher bit rates, B-frames become less useful since they inherently do not contribute to the progressive refinement of an image sequence (i.e. not used as prediction by subsequent coded frames). Regardless, B-frames are still politically controversial.

B pictures are interpolative in two ways: 1. predictions in the bi-directional macroblocks are an average from block areas of two pictures 2. B pictures fill in or interpolate the 3-D video signal over a 33 or 25 millisecond picture period without contributing to the overall signal quality beyond that immediate point in time. In other words, a B pictures, regardless of its internal make-up of macroblock types, has a life limited to its immediate self. As mentioned before, its energy does not propagate into other frames. In a sense, bits spent on B pictures are wasted.

20. Why do some people hate B-frames?

A. Computational complexity, bandwidth, delay, and picture buffer size are the four B-frame Pet Peeves. Computational complexity in the decoder is increased since a some macroblock modes require averaging between two macroblocks.

Worst case, memory bandwidth is increased an extra 15.2 MByte/s (4:2:0 601 rates, not including any half pel or page-mode overhead) for this extra prediction. An extra picture buffer is needed to store the future prediction reference (bi-directionality). Finally, extra delay is introduced in encoding since the frame used for backwards prediction needs to be transmitted to the decoder before the intermediate B-pictures can be decoded and displayed.

Cable television (e.g. -- more like i.e.-- General Instruments) have been particularly adverse to B-frames since, for CCIR 601 rate video, the extra picture buffer pushes the decoder DRAM memory requirements past the magic 8-Mbit (1 Mbyte) threshold into the evil realm of 16 Mbits (2 Mbyte).... although 8-Mbits is fine for 352 x 480 B picture sequence. However, cable often forgets that DRAM does not come in convenient high-volume (low cost) 8-Mbit packages as does the friendly 4-Mbit and 16-Mbit. In a few years, the cost difference between 16 Mbit and 8 Mbit will become insignificant compared to the bandwidth savings gain through higher compression. For the time being, some cable boxes will start with 8-Mbit and allow future drop-in upgrades to the full 16-Mbit.

21. Why was the 16x16 area chosen?

A. The 16x16 area corresponds to the Least Common Multiple (LCM) of 8x8 blocks, given the normative 4:2:0 chroma ratio. Starting with medium size images, the 16x16 area provides a good balance between side information overhead & complexity and motion compensated prediction accuracy. In gist, 16x16 seemed like a good trade-off.

22. Why was the 8x8 DCT size chosen?
A. Experiments showed little improvements with larger sizes vs. the increased complexity. A fast DCT algorithm will require roughly double the arithmetic operations per sample when the transform point size is doubled. Naturally, the best compaction efficiency has been demonstrated using
locally adaptive block sizes (e.g. 16x16, 16x8, 8x8, 8x4, and 4x4) [See Baker and Sullivan]. Naturally, this introduces additional side information

overhead and forces the decoder to implement programmable or hardwired recursive DCT algorithms. If the DCT size becomes too large, then more edges (local discontinuities) and the like become absorbed into the transform block, resulting in wider propagation of Gibbs (ringing) and other phenomena. Finally, with larger transform sizes, the DC term is even more critically sensitive to quantization noise.

23. What is motion compensated prediction, and why is it a pain?

A. MCP in the decoder can be thought of as having four stages:

1. Motion vector computation
2. Prediction retrieval
   various predictions are 16x16, 16x8, 8x4, 8x8 plus any half-pel overhead (e.g. 17x16, 17x17, etc).
3. Filtering
   3.1 Forming half-pel predictions through bi-linear interpolation.
   3.2 Averaging two predictions together (B macroblocks, Dual Prime)
4. Combination and ordering
   4.1 combining 1 or 2 predictions from stage three into upper and lower halves (16 x 8, field in frame)
   4.2 interleaving or grouping together odd and even lines in frame picture predictions.

The final, combined prediction is always a 16x16 block of luminance and 8x8 block of chrominance, just like we experience in MPEG-1.

A single motion vector can be associated with each source, hence a macroblock can have as many as 4 motion vectors.

24. What are the various prediction modes in MPEG-2?

24.1 Frame:
Predictions are formed from a 16 x 16 pixel area in a previously reconstructed frame. Identical to MPEG-1. There can be only one source in forward or backward predicted macroblocks, and two sources in bi-directional macroblocks. The prediction frame itself may have been coded as either a frame or two fields, however once a frame is reconstructed, it is simply a frame as far as future predictions are concerned.

24.2 Field predictions in frame-coded pictures:

Separate predictions are formed for the top (8 lines from field 1)and bottom (8 lines from field 2) portions of the macroblock. A total of two motion vectors in forward or backward predictions, four in bi-directional.

24.3 Field predictions in field-coded pictures:

Predictions are formed from the two most recently decoded fields. Prediction sizes are 16x16, however the 16 lines have a corresponding projection onto a 16x32 pixel area of a frame. One motion vector for forward or backward predictions, and two for bi-directional.

24.4 16x8 predictions in field-coded pictures:

Like field macroblocks in frame-coded pictures, the upper and lower 8 lines in this macroblock mode can have different predictions (hence two motion vectors). This mode compensates for the reduced temporal prediction precision of field picture macroblocks (a result of the fact that fields inherently possess half the number of lines that frames do). The field prediction area projected onto a frame is restored to 16 lines. 2 motion vectors for backwards or forwards, 4 for bi-directional.

24.5 Dual Prime prediction in frame and field-coded pictures

Predictions for the current macroblock are formed from the average of two 16 x 8 line areas from the two most recently decoded fields. Dual Prime was devised as an alternative for B pictures in low delay applications, but still offers many of the signal
quality benefits of B-pictures. Dual Prime requires one less prediction picture buffer, but still retains the same instantaneous prediction bandwidth of a B picture system. As an alternative to coding separate motion vectors for each of the upper and lower 16x8 areas, a full motion vector is sent for

the first area, and a +1, 0, or -1 differential vector (variable length coded) is specified for the second prediction area. A macroblock will have total of two full motion vectors and two differential vectors in frame-coded pictures. Due to the prediction bandwidth overhead, Main Profile restricts the use of Dual Prime prediction to P picture sequences only. High Profile permits use of Dual Prime in B pictures.

24.6 Field and frame organized macroblocks:

Originally intended as a cheaper means of achieving field-decorrelation in frame-coded pictures without the fussy overhead of separate field prediction estimates, the dct coefficients (quantized prediction error for a given macroblock) may be organized into either a field or frame pattern. Essentially this means that the prediction error for the combined 16x16 macroblock may be grouped into field or frame blocks. A bit in the macroblock header (dct_type) indicates whether the upper and lower portions of the macroblock are to be interleaved (frame organized) or remain separated (field organized).

25. How do you tell a MPEG-1 bitstream from a MPEG-2 bitstream?

A. All MPEG-2 bitstreams must contain specific extension headers that *immediately* follow MPEG-1 headers. At the highest layer, for example, the MPEG-1 style sequence_header() is followed by sequence_extension() exclusive to MPEG-2. Some extension headers are specific to MPEG-2 profiles. For example, sequence_scalable_extension() is not allowed in Main Profile bitstreams.

A simple program need only scan the coded bitstream for byte-aligned start codes to determine whether the stream is MPEG-1 or MPEG-2.

26. What is the reasoning behind MPEG syntax symbols?

A. Here are some of the Whys and Wherefores of MPEG symbols:

Start codes
These 32-bit byte-aligned codes provide a mechanism for cheaply searching coded bitstreams for commencement of various layers of video without having to actually parse variable-length codes or perform any decoder arithmetic. Start codes also provide a mechanism for resynchronization in the presence of bit errors.

Coded block pattern (CBP --not to be confused with Constrained Parameters!) When the frame prediction is particularly good, the displaced frame difference (DFD, or prediction error) tends to be small, often with entire block energy being reduced to zero after quantization. This usually happens only at low bit rates. Coded block patterns prevent the need for transmitting EOB symbols in those zero coded blocks.

DCT_coefficient_first
Each intra coded block has a DC coefficient. With coded block patterns signaling all possible combinations of all-zero valued blocks, the dct_coef_first mechanism assigns a different meaning to the VLC codeword that would otherwise represent EOB as the first coefficient.

End of Block:
Saves unnecessary run-length codes. At optimal bitrates, there tends to be few AC coefficients concentrated in the early stages of the zig-zag vector. In MPEG-1, the 2-bit length of EOB implies that there is an average of only 3 or 4 non-zero AC coefficients per block. In MPEG-2 Intra (I) pictures, with a 4-bit EOB code, this number is between 9 and 16 coefficients. Since EOB is required for all coded blocks, its absence can signal that a syntax error has occurred in the bitstream.

Macroblock stuffing
A genuine pain for VLSI implementations, macroblock stuffing was introduced to maintain smoother, constant bitrate control in MPEG-1. However, with normalized complexity measures and buffer management performed a priori (pre-frame, pre-slice, and pre-macroblock) in the

MPEG-2 encoder test model, the need for such localized smoothing evaporated. Stuffing can be achieved through virtually unlimited slice start code padding if required. A good rule of thumb: if you find yourself often using stuffing more than once per slice, you probably don't have a very good rate control algorithm. Anyway, macroblock stuffing is now illegal in MPEG-2, so don t start using it if you already haven t.

MPEG's modified Huffman VLC tables

The VLC tables in MPEG are not Huffman tables in the true sense of Huffman coding, but are more like the tables used in Group 3 fax. They are entropy constrained, that is, non-downloadable and optimized for a limited range of bit rates (sweet spots). With the exception of a few codewords, the larger tables were carried over from the H.261 standard of 1990. MPEG-2 added an "Intra table". Note that the dct_coefficient tables assume positive/negative coefficient pmf symmetry.

27. Why bother to research compressed video when there is a standard?
A. Despite the worldwide standard, many areas remain open for research: advanced encoding and pre-processing, motion estimation, macroblock decision models, rate control and buffer management in editing environments, etc. There's practically no end to it.

28. Where can I get a copy of the latest MPEG-2 draft?

A. Contact your national standards body (e.g. ANSI Sales in NYC for the U.S., British Standards Institute in the UK, etc.). A number of private organizations offer ISO documents.

29. What are the latest working drafts of MPEG-2 ?
A. MPEG-2 has reached voting document of the Draft International Standard for :

Information Technology -- Generic Coding of Moving Pictures and Associated Audio. Recommendation H.262, ISO/IEC Draft International Standard 13818-2. [produced March 25, 1994, not yet approved by voting process].

Audio is Part 1, Video Part 2, and Systems is Part 3. A committee draft for Conformance (Part 4) is expected in Novemeber 1994, as well as the Technical Report on Software Simulation (Part 5).

30. What is the latest version of the MPEG-1 documents?

A. Systems (ISO/IEC IS 11172-1), Video (ISO/IEC IS 11172-2), and Audio (ISO/IEC IS 11172-3) have reached the final document stage. Part 4, Conformance Testing, is currently DIS

31. What is the evolution of ISO standard documents?

A. In chronological order:

ISO/Committee notation	Author's notation
Problem (unofficial first stage)	Barroom Witticism
New work Item (NI)	Napkin Item
New Proposal (NP)	Need Permission
Working Draft (WD)	We're Drunk
Committee Draft (CD)	Calendar Deadlock
Draft International Standard (DIS)	Doesn't Include Substance
International Standard (IS)	Induced patent Statements

32. Where is a good introductory paper to MPEG?

A. Didier Le Gall, "MPEG: A Video Compression Standard for Multimedia Applications," Communications of the ACM, April 1991, Vol.34, No.4, pp. 47-58

33. What are some journals on related MPEG topics ?
A.

IEEE Transactions on Consumer Electronics

IEEE Transactions on Broadcasting
IEEE Transactions on Circuits and Systems for Video Technology
Advanced Electronic Imaging
Electronic Engineering Times (EE Times -- more tabloid coverage. Unfortunate columns by Richard Doherty)
IEEE Int'l Conference on Acoustics, Speech, and Signal Processing (ICASSP)
International Broadcasting Convention (IBC)
Society of Motion Pictures and Television Engineers (SMPTE)
SPIE conference on Visual Communications and Image Processing
SPIE conference on Video Compression for Personal Computers
IEEE Multimedia [first edition Spring 1994]

34. Is there a book on MPEG video?

A. Yes, there will be a book published sometime in 1994 by the same authors who brought you the JPEG book (Bill Pennebaker, Joan Mitchell). Didier Le Gall will be an additional co-author, and will insure digressions into, e.g. arithmetic coding aspects, be kept to a minimum :-)

35. Is it MPEG-2 (Arabic numbers) or MPEG-II (roman)?

A. Committee insiders most often use the Arabic notation with the hyphen, e.g. MPEG-2. Only the most retentive use the official designation: Phase 2. In fact, M.P.E.G. itself is a nickname. The official title is: ISO/IEC JTC1 SC29 WG11. The militaristic lingo has so far managed to keep the enemy (DVI) confused and out of the picture.

ISO: International Organization for Standardization
IEC: International Electrotechnical Commission
JTC1: Joint Technical Committee 1
SC29: Sub-committee 29
WG11: Work Group 11 (moving pictures with... uh, audio)

36. What happened to MPEG-3?

A. MPEG-3 was to have targeted HDTV applications with sampling dimensions up to 1920 x 1080 x 30 Hz and coded bitrates between 20 and 40 Mbit/sec. It was later discovered that with some (compatible) fine tuning, MPEG-2 and MPEG-1 syntax worked very well for HDTV rate video. The key is to maintain an optimal balance between sample rate and coded bit rate.

Also, the standardization window for HDTV was rapidly closing. Europe and the United States were on the brink of committing to analog-digital subnyquist hybrid algorithms (D-MAC, MUSE, et al). European all-digital projects such as HD-DIVINE and VADIS demonstrated better picture quality with respect to bandwidth using the MPEG syntax. In the United States, the Sarnoff/NBC/Philips/Thomson HDTV consortium had used MPEG-1 syntax from the beginning of its all-digital proposal, and with the exception of motion artifacts (due to limited search range in the encoder), was deemed to have the best picture quality of all three digital proponents. HDTV is now part of the MPEG-2 High-1440 Level and High Level toolkit.

37. What is MPEG-4?
A. MPEG-4 targets the Very Low Bitrate applications defined loosely as having sampling dimensions up to 176 x 144 x 10 Hz and coded bit rates between 4800 and 64,000 bits/sec. This new standard would be used, for example, in low bit rate videophones over analog telephone lines.

This effort is in the very early stages. Morphology, fractals, model based, and anal retentive block transform coding are all in the offering. MPEG-4 is now in the application identification phase.

Scaleable modes of MPEG-2

38. What are the scaleable modes of MPEG-2?
A. Scaleable video is permitted only in the High Profiles.

Currently, there are four scaleable modes in the MPEG-2 toolkit. These modes break MPEG-2 video into different layers (base, middle, and high layers) mostly for purposes of prioritizing video data. For example, the high priority channel (bitstream) can be coded with a combination of extra error

correction information and/or increased signal strength (i.e. higher Carrier-to-Noise ratio or lower Bit Error Rate) than the lower priority channel. For example, in HDTV, the high priority bitstream (720 x 480) can be decoded under noise conditions were the lower priority (1440 x 960) cannot. This is part of the "graceful degradation concept. Breaking a video signal into two streams (base and enhancements) has a penalty, however. Usually less than 1.5 dB.

Another purpose of salability is complexity division. A standard TV set need only decode the 720 x 480 channel, thus requiring a less expensive decoder processor than a TV set wishing to display 1440 x 960. This is known as simulcasting.

A brief summary of the MPEG-2 video scalability modes:

Spatial Scalablity-- Useful in simulcasting, and for feasible software decoding of the lower resolution, base layer. This spatial domain method codes a base layer at lower sampling dimensions (i.e. "resolution") than the upper layers. The upsampled reconstructed lower (base) layers are then used as prediction for the higher layers.

Data Partitioning-- Similar to JPEG's frequency progressive mode, only the slice layer indicates the maximum number of block transform coefficients contained in the particular bitstream (known as the "priority break point"). Data partitioning is a frequency domain method that breaks the block of 64 quantized transform coefficients into two bitstreams. The first, higher priority bitstream contains the more critical lower frequency coefficients and side informations (such as DC values, motion vectors). The second, lower priority bitstream carries higher frequency AC data.

SNR Scalability-- Similar to the point transform in JPEG, SNR scalability is a spatial domain method where channels are coded at identical sample rates, but with differing picture quality (achieved through quantization step sizes). The higher priority bitstream contains base layer data that can be added to a lower priority refinement layer to construct a higher quality picture.

Temporal Scalability--- A temporal domain method useful in, e.g., stereoscopic video. The first, higher priority bitstreams codes video at a lower frame rate, and the intermediate frames can be coded in a second bitstream using the first bitstream reconstruction as prediction. In stereoscopic vision, for example, the left video channel can be prediction from the right channel.

Other scalability modes were experimented with in MPEG-2 video (such as Frequency Scalability), but were eventually dropped in favor of methods that demonstrated comparable or better picture quality with greater simplicity.

39. Why MPEG-2? Wasn't MPEG-1 enough?

A. MPEG-1 was optimized for CD-ROM or applications at about 1.5 Mbit/sec. Video was strictly non-interlaced (i.e. progressive). The international cooperation executed well enough for MPEG-1, that the committee began to address applications at broadcast TV sample rates using the CCIR 601 recommendation (720 samples/line by 480 lines per frame by 30 frames per second or about 15.2 million samples/sec including chroma) as the reference.

Unfortunately, today's TV scanning pattern is interlaced. This introduces a duality in block coding: do local redundancy areas (blocks) exist exclusively in a field or a frame.(or a particle or wave) ? The answer of course is that some blocks are one or the other at different times, depending on motion activity. The additional man years of experimentation and implementation between MPEG-1 and MPEG-2 improved the method of block-based transform coding.

40. What did MPEG-2 add to MPEG-1 in terms of syntax/algorithms ?
A. Here is a brief summary:

Sequence layer:
More aspect ratios. A minor, yet necessary part of the syntax.

Horizontal and vertical dimensions are now required to be a multiple of 16 in frame coded pictures, and the vertical dimension must be a multiple of 32 in field coded pictures.

4:2:2 and 4:4:4 macroblocks were added in the Next profiles.

Syntax can now signal frame sizes as large as 16383 x 16383.

Syntax signals source video type (NTSC, PAL, SECAM, MAC, component) to help post-processing and display.

Source video color primaries (609, 170M, 240M, D65, etc.) and opto-electronic transfer characteristics (709, 624-4M, 170M etc.) can be indicated.

Four scaleable modes [see scalability discussion]

Picture layer:
All MPEG-2 motion vectors are specified to a half-pel sample grid.

DC precision can be user-selected as 8, 9, 10, or 11 bits.

New scalar quantization matrices may be downloaded once per picture. In High profile, separate chrominance matrices now exist (Y and C no longer have to share)

Concealment motion vectors were added to I-pictures in order to increase robustness from bit errors. I pictures are the most critical and sensitive picture in a group of pictures.

A non-linear macroblock quantization factor providing a wider dynamic range, from 0.5 to 56, than the linear MPEG-1 (1 to 32) range. Both are sent as a 5-bit FLC side information in the macroblock and slice headers.

New Intra-VLC table for dct_coefficient_next (AC run-level events) that is a better match for the histogram of Intra-coded pictures. EOB is 4 bits. The old table, dct_coef_next, are reserved for use in non-intra pictures (P, B), although they new table can be used for Intra-coded macroblocks in P and B pictures as well.

Alternate scanning pattern that (supposedly) improves entropy coding performance over the original Zig-Zag scan used in H.261, JPEG, and MPEG-1. The extra scanning pattern is geared towards interlaced video.

Syntax to signal an irregular 3:2 pulldown process (repeat_field_first flag)

Progressive and interlaced frame coding

Syntax to indicate source composite video characteristics useful in post-processing operations. (v-axis, field sequence, sub_carrier, phase, burst_amplitude, etc.)

Pan & scanning syntax that tells decoder how to, for example, window a 4:3 image within a wider 16:9 aspect ratio coded image. Vertical pan offset has 1/16th pixel accuracy.

Macroblock layer:
Macroblock stuffing is now illegal in MPEG-2 (hurray!!). If stuffing is really needed, the encoder can pad slice start codes.

Two organizations for macroblock coefficients (interlaced and progressive) signaled by dct_type flag.

Now only one run-level escape code code (24-bits) instead of the single (20-bits) and double escape (28-bits) in MPEG-1.

Improved mismatch control in quantization over the original oddification method in MPEG-1. Now specifies adding or subtracting one to the 63rd AC coefficient depending on parity of the summed coefficients. MPEG-2

mismatch control is performed on the transform coefficients, whereas in MPEG-1, it is applied to the quantized transform coefficients.

Many additional prediction modes (16x8 MC, field MC, Dual Prime) and, correspondingly, macroblock modes.

Overall, MPEG-2's greatest compression improvements over MPEG-1 are: prediction modes, Intra VLC table, DC precision, non-linear macroblock quantization. Implementation improvements: macroblock stuffing was eliminated.

41. How do MPEG and JPEG differ?

A. The most fundamental difference is MPEG's use of block-based motion compensated prediction (MCP)---a method falling into the general category of temporal DPCM.

The second most fundamental difference is in the target application. JPEG adopts a general purpose philosophy: independence from color space (up to 255 components per frame) and quantization tables for each component. Extended modes in JPEG include two sample precision (8 and 12 bit sample accuracy), combinations of frequency progressive, spatial hierarchically progressive, and amplitude (point transform) progressive scanning modes. Further color independence is made possible thanks to downloadable Huffman tables (up to one for each component.)

Since MPEG is targeted for a set of specific applications, there is only one color space (4:2:0 YCbCr), one sample precision (8 bits), and one scanning mode (sequential). Luminance and chrominance share quantization and VLC tables. MPEG adds adaptive quantization at the macroblock (16 x 16 pixel area) layer. This permits both smoother bit rate control and more perceptually uniform quantization throughout the picture and image sequence. However, adaptive quantization is part of the Enhanced JPEG charter (ISO/IEC 10918-3) currently in verification stage. MPEG variable length coding tables are non-downloadable, and are therefore optimized for a limited range of compression ratios appropriate for the target applications.

The local spatial decorrelation methods in MPEG and JPEG are very similar. Picture data is block transform coded with the two-dimensional orthanormal 8x8 DCT, with asymmetric basis vectors about time (aka DCT-II). The resulting 63 AC transform coefficients are mapped in a zig-zag pattern (or alternative scan pattern in MPEG-2) to statistically increase the runs of zeros. Coefficients of the vector are then uniformly scalar quantized, run-length coded, and finally the run-length symbols are variable length coded using a canonical (JPEG) or modified Huffman (MPEG) scheme. Global frame redundancy is reduced by 1-D DPCM of the block DC coefficients, followed by quantization and variable length entropy coding of the quantized DC coefficient.

```
            MCP                    DCT                     ZZ
Q
  Frame -> 8x8 spatial block -> 8x8 frequency block -> Zig-zag scan ->

                    RLC                    VLC
       quanitzation -> run-length coding -> variable length coding.
```

The similarities have made it possible for the development of hard-wired silicon that can code both standards. Even some highly microcoded architectures employing hardwired instruction primitives or functional blocks benefit from JPEG/MPEG similarities. There are many additional yet minor differences. They include:

1. In addition to the 8-bit mode, DCT and quantization precision in MPEG has a 9-bit and 12-bit mode, respectively, exclusively in non-intra coded macroblocks. A 1-bit expansion takes place in the macroblock difference operation.

2. Mismatch control in MPEG-1 forces quantized coefficients to become odd values (oddification). JPEG does not employ any mismatch mechanism.

3. JPEG run-length coding produces run-size tokens (run of zeros,

non-zero coefficient magnitude) whereas MPEG produces fully concatenated run-level tokens that do not require magnitude differential bits.

4. DC values in MPEG-1 are limited to 8-bit precision (a constant stepsize of 8), whereas JPEG DC precision can occupy all possible 11-bits. MPEG-2, however, re-introduced extra DC precision critical even at high compression ratios.

Difference between MPEG and H.261

42. How do MPEG and H.261 differ?

A. H.261, also known as Px64, was targeted for teleconferencing applications where motion is naturally more limited. Motion vectors are restricted to a range of +/- 15 pixel unit displacements. Prediction accuracy is reduced since H.261 motion vectors are specified to only integer-pel accuracy. Other quality syntactic differences include: no B-pictures, inferior mismatch control.

43. Is H.261 the de facto teleconferencing standard?

A. Not exactly. To date, about seventy percent of the industrial teleconferencing hardware market is controlled by PictureTel of Mass. The second largest market controller is Compression Labs of Silicon Valley. PictureTel hardware includes compatibility with H.261 as a lowest common denominator, but when in communication with other PictureTel hardware, it can switch to a mode superior at low bit rates (less than 300kbits/sec). In fact, over 2/3 of all teleconferencing is done at two-times switched 56 channel (~P = 2) bandwidth. ISDN is still expensive. In each direction, video and audio are coded at an aggregate rate of 112 kbits/sec (2*56 kbits/sec). The PictureTel proprietary compression algorithm is acknowledged to be a combination of spatial pyramid, lattice vector quantizer, and an unidentified entropy coding method. Motion compensation is considerably more refined and sophisticated than the 16x16 integer-pel block method specified in H.261.

The Compression Labs proprietary algorithm also offers significant improvement over H.261 when linked to other CLI hardware. Local decorrelation is based on a DCT-VQ hybrid.

Currently, ITU-TS (International Telecommunications Union--teleconferencing Sector), formerly CCITT, is quietly defining an improvement to H.261 with the participation of industry vendors.

Rate control

44. What is the TM rate control and adaptive quantization technique ?

A. The Test model (MPEG-2) and Simulation Model (MPEG-1) were not, by any stretch of the imagination, meant to epitomize state-of-the art encoding quality. They were, however, designed to exercise the syntax, verify proposals, and test the *relative* compression performance of proposals in a timely manner that could be duplicated by co-experimenters. Without simplicity, there would have been no doubt endless debates over model interpretation. Regardless of all else, more advanced techniques would probably trespass into proprietary territory.

The final test model for MPEG-2 is TM version 5b, aka TM version 6. The final MPEG-1 simulation model is version 3. The MPEG-2 TM rate control method offers a dramatic improvement over the SM method. TM adds more accurate estimation of macroblock complexity through use of limited a priori information. Macroblock quantization adjustments are computed on a macroblock basis, instead of once-per-slice.

45. How does the TM work?
A. Rate control and adaptive quantization are divided into three steps:

Step One:Bit Allocation

In Complexity Estimation, the global complexity measures assign

relative weights to each picture type (I,P,B). These weights (Xi, Xp, Xb) are reflected by the typical coded frame size of I, P, and B pictures (see typical frame size discussion). I pictures are usually assigned the largest weight since they have the greatest stability factor in an image sequence. B pictures are assigned the smallest weight since B energy do not propagate into other pictures and are usually highly correlated with neighboring P and I pictures.

The bit target for a frame is based on the frame type, the remaining number of bits left in the Group of Pictures (GOP) allocation, and the immediate statistical history of previously coded pictures.

Step Two: Rate Control

Rate control attempts to adjust bit allocation if there is significant difference between the target bits (anticipated bits) and actual coded bits for a block of data. If the virtual buffer begins to overflow, the macroblock quantization step size is increased, resulting in a smaller yield of coded bits in subsequent macroblocks. Likewise, if underflow begins, the step size is decreased. The Test Model approximates that the target
picture has spatially uniform distribution of bits. This is a safe approximation since spatial activity and perceived quantization noise are almost inversely proportional. Of course, the user is free to design a custom distribution, perhaps targeting more bits in areas that contain text, for example.

Step Three: Adaptive Quantization

The final step modulates the macroblock quantization step size obtained in Step 2 by a local activity measure. The activity measure itself is normalized against the most recently coded picture of the same type (I, P, or B). The activity for a macroblock is chosen as the minimum among the four 8x8 block luminance variances. Choosing the minimum block is part of the concept that a macroblock is no better than the block of highest visible distortion (weakest link in the chain).

46. What is a good motion estimation method, then?

A. When shopping for motion vectors, the three basic characteristics are: Search range, search pattern, and matching criteria. Search pattern has the greatest impact on finding the best vector. Hierarchical search patterns first find the best match between downsampled images of the reference and target pictures and then refine the vector through progressively higher resolutions. When compared to other fast methods, hierarchical patterns are less likely to be confused by extremely local distortion minimums as being a best match. Also note that subsampled search and hierarchical search are not synonymous.

Q. Is there a limit to the length of motion vectors?

The search area is unlimited, but the reconstructed motion vectors must not:

a. point beyond the picture boundaries (1 <= MV_x <= luminancewidth - 16) and (1 <= MV_y <= luminanceheight - 16). The - 16 is due to the fact that the motion vector origin is the upper left hand corner of a macroblock)

b. In Constrained Parameters MPEG-1, the motion vector is limited to a range of [-64,+63.5] luminance samples with half-pel accuracy, and [-128,+127.5] with integer pel accuracy. Break the constrained parameters rules and your video sequence will not likely display on many hardware devices.

c. In MPEG-2 Video Main Profile at Main Level, the motion vectors are always on a half-pel co-ordinate grid, and the vertical range is restricted to [-64, +63.5], and the horizontal limit is [-256,+255.5].

d. in MPEG-1, the syntactic limit of the motion vector is [-1024,+1023] integer pel, horizontal and vertical.

e. in MPEG-2, the syntactic limit of the motion vector is [-2048,+2047.5] horizontal, [-1024,+1023.5] vertical.

47. Is exhaustive search "optimal" ?

A. Definitely not in the context of block-based MCP video. Since one motion vector represents the prediction of 256 pixels, divergent pixels within the macroblock are misrepresented by the "global" vector. This leads back to the general philosophy of block-based coding as an approximation technique. In their ICASSP'93 paper, Sullivan discusses ways in which block-based prediction schemes can solve part of this problem.

Exhaustive search may find blocks with the least distortion (displaced frame difference) but will not produce motion vectors with the lowest entropy.

48. What are some advanced encoding methods?

Quantizer feedback: determine the dependent quantization stepsize by modeling quantization error propagating over multiple pictures. [Uz/et al ICASSP 93, Ortega/Vetterli/et al ICASSP 93]

Smoothness constraint placed on local activity measures. immediate blocks outside target macroblock are considered when selecting macroblock quantization stepsize .[Thomson/Savitier patent]

Horizontal variance: measure variance between columns of pixels in addition to the traditional measure of variance along rows (lines) when making field/frame macroblock prediction decision.

DFD energy: examine DFD energy/variance when making Intra/Non-intra macroblock decision.

Activity measures: use total bits from a first-pass encoding of a picture or macroblock as a measure of the activity. Coded bits is a more accurate reflection of local complexity than variance. [Thomson/Savitier patent]

motion vector cost: this is true for any syntax elements, really. Signaling a macroblock quantization factor or a large motion vector differential can cost more than making up the difference with extra quantized DFD (prediction error) bits. The optimum can be found with, some Lagrangian operator. In summary, any compression system with side information, there is a optimum point between signaling overhead (e.g. prediction) and prediction error.

Liberal Interpretations of the Forward DCT:
Borrowing from the concept that the DCT is simply a filter bank, a technique that seems to be gaining popularity is basis vector shaping. Usually this is combined with the quantization stage since the two are tied closely together in a rate-distortion sense. The idea is to use the basis vector shaping as a cheap alternative to pre-filtering by combining the more desirable data adaptive properties of pre-filtering/ pre-processing into the transformation process... yet still reconstruct a picture in the decoder using the standard IDCT that looks reasonably like the source. Some more clever schemes will apply a form of windowing. [Warning: watch out for eigenimage/basis vector orthoganality.]

Frequency-domain enhancements:
Enhancements are applied after the DCT (and possibly quantization)stage to the transform coefficients. This borrows from the concept: if you don't like the (quantized) transformed results, simply reshape them into something you do like. Suppressing isolated small amplitudes is popular.

Temporal spreading of quantization error:
This method is similar to the original intent behind color subcarrier phase alternation by field in the NTSC, PAL, and SECAM analog TV standards: for stationary areas, noise does not hang" in one location, but dances about the image over time to give a more uniform effect. Distribution makes it more difficult for the eye to "catch on" to trouble spots (due to the latent temporal response curve of human vision). Simple encoder models tend to do this naturally but will not solve all situations.

Look-ahead and adaptive frame cycle structures: analyze picture activity several pictures into the future, looking for scene changes or motion statistics.

It is easy to spot encoders that do not employ any advanced encoding techniques: reconstructed video usually contains ringing around edges, color bleeding, and lots of noise.

49. Is so-and-so really MPEG compliant ?

A. At the very least, there are two areas of conformance/compliance in MPEG: 1. Compliant bitstreams 2. compliant decoders. Technically speaking, video bitstreams consisting entirely of I-frames (such as those generated by Xing software) are syntactically compliant with the MPEG specification. The I-frame sequence is simply a subset of the full syntax. Compliant bitstreams must obey the range limits (e.g. motion vectors limited to +/-128, frame sizes, frame rates, etc.)and syntax rules (e.g. all slices must commence and terminate with a non-skipped macroblock, no gaps between slices, etc.).

Decoders, however, cannot escape true conformance. For example, a decoder that cannot decode P or B frames are *not* legal MPEG. Likewise, full arithmetic precision must be obeyed before any decoder can be called "MPEG compliant." The IDCT, inverse quantizer, and motion compensated predictor must meet the specification requirements... which are fairly rigid (e.g. no more than 1 least significant bit of error between reference and test decoders). Real-time conformance is more complicated to measure than arithmetic precision, but it is reasonable to expect that decoders that skip frames on reasonable bitstreams are not likely to be considered compliant.

Artifacts

50. What are the tell-tale MPEG artifacts?

A. If the encoder did its job properly, and the user specified a proper balance between sample rate and bitrate, there shouldn't be any visible artifacts. However, in sub-optimal systems, you can look for:

Gibbs phenomenon/Ringing/Aliasing (too few AC bits, not enough pre-processing)

Blockiness (not considering your neighbors before quantizing)

Posterization (too few DC bits)

Checkerboards (DCT eigenimages as a result of too few AC coefficients)
Colorbleeding (not considering color in encoder cost model, not subtracting color at edges of objects, etc.)

51. Where are the weak points of MPEG video ?
A.
Texture patterns (rapidly alternating lines)
sharp edges (especially text)
[installment 3]

52. What are some myths about MPEG?
A. There are a few major myths that I am aware of:

1. Block displacements: macroblock predictions are formed out of arbitrary 16x16 (or 16x8/16x16 in MPEG-2) areas from previously reconstructed pictures. Many people believe that the prediction macroblocks have boundaries that fall on interchange boundaries (pixel 0, 15, 31, 53... line 0, 15, 31, 53... etc.). In fact, motion vectors represent relative translations with respect to the target reconstruction macroblock coordinates. The motion vectors can point to half pixel coordinates, requiring that the prediction macroblock to be formed via bi-linear interpolation of pixels.

2. Displaced frame (macroblock) difference construction: the prediction error formed as the difference between the prediction macroblock and

source macroblock is coded much like an Intra macroblock. The prediction may come from different locations (as in bi-directional prediction--or in MPEG-2--16x8, field-in-frame, and Dual Prime), but the DFD is always coded as a 16x16 unit.

3. Compression ratios

You hear 200:1 and 100:1 in the media. Utter rubbish. The true range is between 16:1 and 40:1. Spreading misinformation about compression ratios in public will catch the attention of the infamous MPEG Police. They say mild-mannered Michael Barnsley will snap, without warning, into violent rage if he doesn't get the upper bunk bed.

4. Picture coding types all consist of the same macroblocks

Macroblocks within I pictures are strictly intra-coded. Macroblocks within P pictures can be either predicted or intra-coded, and B pictures they can be bi-directional, forward, backward, or intra. Additional macroblock modes switches include: predicted with no motion compensation, modified macroblock quantization, coding of prediction error or not. The switches are concatenated into the macroblock_type side information and variable length coded in the macroblock header.

53. What is the color space of MPEG?

MPEG strictly specifies the YCbCr color space, not YUV or YIQ or YPbPr or YDrDb or any other color difference variations. Regardless of any bitstream parameters, MPEG-1 and MPEG-2 Video Main Profile specify 4:2:0 chroma ratio, where the color difference channels (Cb, Cr) have half the resolution or sample grid density in both the horizontal and vertical direction
with respect to luminance.

MPEG-2 High Profile includes an option for 4:2:2 and 4:4:4 coding. Applications
for this are likely to be broadcasting and contribution equipment.

54. Don't you mean 4:1:1 ?

A. No, here is a table of ratios:

format	CCIR 601 (60 Hz) image Y	Cb, Cr	Chroma sub-sampling factors Vertical	Horizontal
4:4:4	720 x 480	720 x 480	none	none
4:2:2	720 x 480	360 x 480	none	2:1
4:2:0	720 x 480	360 x 240	2:1	2:1
4:1:1	720 x 480	720 x 120	none	4:1
4:1:0	720 x 480	180 x 120	4:1	4:1

3:2:2, 3:1:1, and 3:1:0 are less common variations.

55. Why did MPEG choose 4:2:0 ? Isn't 4:2:2 the standard for TV?

A. At least three reasons I can think of:

1. 4:2:0 picture memory requirements are 33% less than the size of 4:2:2 pictures.
MPEG-1 decoder are able to snugly fit all 3 SIF pictures (1 reconstruction & display, 2 prediction) into 512 KBytes of buffer space. CCIR 601 is a tighter fit into 2 Mbytes.

2. The subjective difference between 4:2:0 and 4:2:2 is minimal, when considering consumer display equipment and distribution compression ratios.

3. Vertical decimation increases compression efficiency by reducing syntax overhead posed in an 8 block (4:2:0) macroblock structure.

4. You re compressing the hell out of the video signal, so what possible difference can the 0:0:2 high-pass make?

Interlacing and the 62 microsecond gap between successively scanned lines

introduces some discontinuities, but most of this can be alleviated through pre-processing.

56. What is the precision of MPEG samples?

A. By definition, MPEG samples have no more and no less than 8-bits uniform sample precision (256 quantization levels). For luminance (which is unsigned) data, black corresponds to level 0, white is level 255. However, in CCIR recommendation 601 chromaticy, levels 0 through 14 and 236 through 255 are reserved for blanking signal excursions. MPEG currently has no such clipped excursion restrictions, although decoder might take care to insure active samples do not exceed these limits. With three color components per pixel, the total combination is roughly 16.8 million colors (i.e. 24-bits).

57. What is all the fuss with cositing of chroma components?

A. It is moderately important to properly co-site chroma samples, otherwise a sort of chroma shifting effect (exhibited as a halo) may result when the reconstructed video is displayed. In MPEG-1 video, the chroma samples are exactly centered between the 4 luminance samples (Fig 1.) To maintain compatibility with the CCIR 601 horizontal chroma locations and simplify implementation (eliminate need for phase shift), MPEG-2 chroma samples are arranged as per Fig.2.

```
  Y   Y   Y   Y            Y   Y   Y   Y         YC  Y   YC  Y
    C       C              C       C
  Y   Y   X   Y            Y   Y   Y   Y         YC  Y   YC  Y

  Y   Y   Y   Y            Y   Y   Y   Y         YC  Y   YC  Y
    C       C              C       C
  Y   Y   Y   Y            Y   Y   Y   Y         YC  Y   YC  Y

  Fig.1 MPEG-1              Fig.2  MPEG-2           Fig.3 MPEG-2 and
 4:2:0 organization        4:2:0 organization         CCIR Rec. 601
                                                    4:2:2 organization
```

MPEG for the data compression expert

58. How would you explain MPEG to the data compression expert?

A. MPEG video is a block-based video scheme.

59. How does MPEG video really compare to TV, VHS, laserdisc ?
A. VHS picture quality can be achieved for source film video at about 1 million bits per second (with proprietary encoding methods). It is very difficult to objectively compare MPEG to VHS. The response curve of VHS places -3 dB at around 2 MHz of analog luminance bandwidth (equivalent to 200 samples/line). VHS chroma is considerably less dense in the horizontal direction than MPEG source video (compare 80 samples/line to 176!). From a sampling density perspective, VHS is superior only in the vertical direction (480 luminance lines compared to 240)...
but when taking into account (supposedly such things as) interfield magnetic tape crosstalk and the TV monitor Kell factor, the perceptual vertical advantage is not all that significant. VHS is prone to such inconveniences as timing errors (an annoyance addressed by time base correctors), whereas digital video is fully discretized. Pre-recorded VHS is typically recorded at very high duplication speeds (5 to 15 times real time playback speed), opening up additional avenues for artifacts. In gist, MPEG-1 at its nominal parameters can match VHS's sexy low-pass-filtered look.

With careful coding schemes, broadcast NTSC quality can be approximated at about 3 Mbit/sec, and PAL quality at about 4 Mbit/sec. Of course, sports sequences with complex spatial-temporal activity should be treated with bit rates more like 5 and 6 Mbit/sec, respectively. Laserdisc is a tough one to compare. Laserdisc's are encoded with composite video (NTSC or PAL). Manufacturers of laser disc players make claims of up to 425 TVL (or 567 samples/line) response. Thus it could be said the laserdisc has a 567 x 480 x 30 Hz "potential resolution". The carrier-to-noise ratio is typically better than 48 dB. Timing is excellent. Yet some of the clean characteristics of laserdisc can be achieved with MPEG-1 at 1.15 Mbit/sec (SIF rates), especially for those areas of medium detail (low spatial activity) in the

presence of uniform motion. This may be why some people say MPEG-1 video at 1.15 Mbit/sec looks almost as good as Laserdisc or Super VHS at times.

60. What are the typical MPEG-2 bitrates and picture quality?

	Picture type			
	I	P	B	Average
MPEG-1 SIF @ 1.15 Mbit/sec	150,000	50,000	20,000	38,000
MPEG-2 601 @ 4.00 Mbit/sec	400,000	200,000	80,000	130,000

Note: parameters assume Test Model for encoding, I frame distance of 15 (N = 15), and a P frame distance of 3 (M = 3).

Of course, among differing source material, scene changes, and use of advanced encoder models... these numbers can be significantly different.

61. At what bitrates is MPEG-2 video optimal?
A. The Test subgroup has defined a few examples:

"Sweet spot" sampling dimensions and bit rates for MPEG-2:

Dimensions	Coded rate	Comments
352x480x24 Hz (progressive) (better)	2 Mbit/sec	Half horizontal 601. Looks almost NTSC broadcast quality, and is a good substitute for VHS. Intended for film src.
544x480x30 Hz capture (interlaced)	4 Mbit/sec	PAL broadcast quality (nearly full of 5.4 MHz luminance carrier). Also 4:3 image dimensions windowed within 720 sample/line 16:9 aspect ratio via pan&scan.
704x480x30 Hz (interlaced)	6 Mbit/sec	Full CCIR 601 sampling dimensions.

[these numbers subject to change at whim of MPEG Test subgroup]

62. Why does film perform so well with MPEG ?
A. Several reasons, really:

1) The frame rate is 24 Hz (instead of 30 Hz) which is a savings of some 20%.
2) the film source video is inherently progressive. Hence no fussy interlaced spectral frequencies.
3) the pre-digital source was severely oversampled (compare 352 x 240 SIF to 35 millimeter film at, say, 3000 x 2000 samples). This can result in a very high quality signal, whereas most video cameras do not oversample, especially in the vertical direction.
4) Finally, the spatial and temporal modulation transfer function (MTF) characteristics (motion blur, etc) of film are more amenable to the transform and quantization methods of MPEG.

63. What is the best compression ratio for MPEG ?

A. The MPEG sweet spot is about 1.2 bits/pel Intra and .35 bits/pel inter. Experimentation has shown that intra frame coding with the familiar DCT-Quantization-Huffman hybrid algorithm achieves optimal performance at about an average of 1.2 bits/sample or about 6:1 compression ratio. Below this point, artifacts become noticeable.

64. Can MPEG be used to code still frames?

A. Yes. There are, of course, advantages and disadvantages to using MPEG over JPEG:

Disadvantages:

1. MPEG has only one color space
2. MPEG-1 and MPEG-2 Main Profile luma and chroma share quanitzation and VLC tables
3. MPEG-1 is syntactically limited to 4k x 4k images, and 16k x 16k for MPEG-2.

Advantages:

1. MPEG possesses adaptive quantization

2. With its limited still image syntax, MPEG averts any temptation to use unnecessary, expensive, and academic encoding methods that have little impact on the overall picture quality (you know who you are).

Philips' CD-I spec. has a requirement for a MPEG still frame mode, with double SIF image resolution. This is technically feasible mostly thanks to the fact that only one picture buffer is needed to decode a still image instead of three buffers.

65. Is there an MPEG file format?

A. Not exactly. The necessary signal elements that indicate image size, picture rate, aspect ratio, etc. are already contained within the sequence layer of the MPEG video stream. The Whitebook format for Karoke and CD-I movies specify a range of (time-division) multiplexing strategies for audio and video bitstreams. A directory format listing scenes and their locations on the disc is associated with the White Book specification.

66. What are some pre-processing enhancements ?

Adaptive de-interlacing:

This method maps interlaced video from a higher sampling rate (e.g 720 x 480) into a lower rate, progressive format (352 x 240). The most basic algorithm measures the correlation between two immediate macroblock fields, and if the correlation is high enough, uses an average of both fields to form a frame macroblock. Otherwise, a field area from one field (usually of the same parity) is selected. More clever algorithms are much more complex than this, and may involve median filtering, and multirate/multidimensional tools.

Pre-anti-aliasing and Pre-blockiness reduction:
A common method in still image coding is to pre-smooth the image before encoding. For example, if pre-analysis of a frame indicates that serious artifacts will arise if the picture were to be coded in the current condition (i.e. below the sweet spot), a pre-anti-aliasing filter can be applied. This can be as simple as having a smoothing severity proportional to the image activity. The pre-filter can be global (same smoothing factor for whole image or sequence) or locally adaptive. More complex methods will again use multirate/multidimensional methods.

One straightforward concept from multidimensional/multirate e-processing is to apply source video whose resolution (sampling density) is greater than the target source and reconstruction sample rates. This follows the basic principles of oversampling, as found in A/D converters.

These filters emphasize the fact that most information content is contained in the lower harmonics of a picture anyway. VHS is hardly considered to be a sharp cut-off medium, tragically implying that "320 x 480 potential" of VHS is never truly realized.

67. Why use these "advanced" pre-filtering techniques?

A. Think of the DCT and quantizer as an A/D converter. Think of the DCT/Q pre-filter as the required anti-alias prefilter found before every A/D. The big difference of course is that the DCT quantizer assigns a varying number of bits per transform coefficient. Judging on the normalized activity measured in the pre-analysis stage of video encoding (assuming you even have

a pre-analysis stage), and the target buffer size status, you have a fairly good idea of how many bits can be spared for the target macroblock, for example.

Other pre-filtering techniques mostly take into account: texture patterns, masking, edges, and motion activity. Many additional advanced techniques can be applied at different immediate layers of video encoding (picture, slice, macroblock, block, etc.).

68. What about post-processing enhancements?

Some research has been carried out in this area. Non-linear interpolation methods have been published by Wu and Gersho (e.g. ICASSP 93), convex hull projections for MAP (Severinson, ICASSP 93), and others. Post-processing unfortunately defies the spirit of MPEG conformance. Decoders should produce similar reconstructions. Enhancements should ideally be done during the pre-processing and encoding stages.

69. Can motion vectors be used to measure object velocity?

A. Motion vector information cannot be reliably used as a means of determining object velocity unless the encoder model specifically set out to do so. First, encoder models that optimize picture quality generate vectors that typically minimize prediction error and, consequently, the vectors often do not represent true object translation. Standards converters that resample one frame rate to another (as in NTSC to PAL) use different methods (motion vector field estimation, edge detection, et al) that are
not concerned with optimizing ratios such as SNR vs bitrate. Secondly, motion vectors
are not transmitted for all macroblocks anyway.

70. How do you code interlaced video with MPEG-1 syntax?
A. Two methods can be applied to interlaced video that maintain syntactic compatibility with MPEG-1 (which was originally designed for progressive frames only). In the field concatenation method, the encoder model can carefully construct predictions and prediction errors that realize good compression but maintain field integrity (distinction between adjacent fields of opposite parity). Some pre-processing techniques can also be applied to the interlaced source video that would, e.g., lessen sharp vertical frequencies.

This technique is not efficient of course. On the other hand, if the original source was progressive (e.g. film), then it is more trivial to convert the interlaced source to a progressive format before encoding. (MPEG-2 would then only offer superior performance through greater DC block precision, non-linear mquant, intra VLC, etc.) Reconstructed frames are re-interlaced in the decoder Display process.

The second syntactically compatible method codes fields as separate pictures. This approach has been acknowledged not to work as well.

71. Is MPEG patented?
A. Yes and no. Many encoding methods are patented. Approximately 11 blocking patents, that is, patents that are general enough to be unavoidable in any implementation have been recently identified.

A patent pool is being formed within MPEG where a single royalty fee would be split among the 31 patent-holding companies.

72. How many cable box alliances are there?

A. Many. To start with:

Scientific Atlanta (SA), Kaledia, and Motorola:
SA will build the box, Motorola the chips, and Kaleida the
O/S and user interface (using ScriptX of course).

Silicon Graphics (SGI), Scientific Atlanta, and Toshiba
For the Time Warner's Orlando trial, SGI will provide the
RISC (MIPS R4000) and software, SA will do the box again,
and Toshiba will provide the chips.

General Instruments (GI) and Microsoft:
GI will make the box and Intel will supply the special low-cost 386SL processor on which a 1MB flash EPROM executable core of Microsoft windows and DOS will run. Microsoft will develop the user interface.

Hewlett Packard (HP):
HP will manufacture and/or design low cost, open architecture set-top decoder boxes (not a part of the Eon wireless deal). The CPU will explicitly not use a 80x68 based processor.

CLI and Philips:
Compression Labs will provide the encoder technology and Philips will provide the decoder techology for an ADSL system whose transport structure will be put together by Broadband Technologies.

["These alliances subject to change at the whim of PR departments and market forces."]

73. Will there be an MPEG video tape format?

A. Not exactly. A consortium of international companies are co-developing a consumer digital video 6 millimeter wide, metal particle tape format. Due to the initial high cost of MPEG encoders, a JPEG-like compression method will be used for inexpensive encoding of typical consumer source video (broadcast PAL, NTSC). The natural consequence of still image methods is less efficient use of bandwidth: 25 Mbit/sec for the same subjective real-time playback quality achieved at 6 Mbit/sec possible with MPEG-2. A second bit rate mode, 50 Mbit/sec, is designated for HDTV.

Pre-coded digital video from, e.g., broadcast sources will be directly recorded to tape and "passed-through" as a coded bitstream to the video decompression box upon tape playback. Assuming if linear tape speed is to be proportional to bit rate, the recording time of a pre-compressed MPEG-2 program at the upper limit of 5 Mbit/sec for broadcast quality video, the recording time would be over 20 hours. Channel coding schemes (error correction, convolution coding, etc.), however, will most likely be optimized for the tape medium and therefore may differ from the channel methods for cable, terrestrial, and satellite. (A Zenith-Goldstar S-VHS based experiment did, however, directly record the 4-VSB broadcast baseband signal of the old Zenith/AT&T HDTV proposal).

More specs: (Summarized from EE Times July 5, 1993 article)

tape width: 6.35 mm
Audio: two channel 48 KHz 16-bit audio, or 4 channel at 32 KHz at 12-bit
Tape format: metal evaporated tape, 13.5 microns thick

Cassette dimensions: (millimeters) Recording times:

Size	Width	Height	Depth	525/625 (25Mb/sec)	HDTV (50 Mb/s)
Standard	125	78	14.6	4h30min	2h15min
Small	66	48	12.2	1 hour	30min

Linear tape speeds: 18.812 mm/s (60Hz), 18.831 mm/s (50 Hz)
Video compression: DCT based

Participants: Matsushita, Sony, Philips, Thomson, Hitachi, Mitsubishi, Sanyo, Sharp, Toshiba, JVC.

MPEG in everyday life

74. Where will be see MPEG in everyday life?
A. Just about wherever you see video today.

DBS (Direct Broadcast Satellite)
The Hughes/USSB DBS service will use MPEG-2 video and audio. Thomson has exclusive rights to manufacture the decoding boxes for the first 18 months of operation. Hughes/USSB DBS will begin its U.S. service in April 1994. Two satellites at 101 degrees West will share the power

requirements of 120 Watts per 27 MHz transponder over a total of 32 transponders. Multi source channel rate control methods will be employed to optimally allocate bits between several programs normalized to one 22 Mbit/sec data carrier. Bit allocation adapts to instantaneous co-channel
spatial and co-channel temporal activity. An average of 150 channels are planned with the addition of a second set of satellites augmenting the power level of each transponder to 240 Watts. The coded throughput of each transponder will increase to 30 Mbit/sec.

CATV (Cable Television)
Despite conflicting options, the cable industry has more or less settled on MPEG-2 video. Audio is less than settled. For example, General Instruments (the largest U.S. consumer cable set-top box manufacturer) have announced the planned exclusive use of Dolby AC-3. The General Instruments DigiCipher I video syntax is similar to MPEG-2 syntax, but employs smaller macroblock predictions and no B-frames. The DigiCipher II specification will include modes to support both the GI and full MPEG-2 Video Main Profile syntax. Digicipher-I services such as HBO will upgrade to DigiCipher II in 1994.

HDTV
The U.S. Grand Alliance, a consortium of companies that formerly competed to win the U.S. terrestrial HDTV standard, have already agreed to use the MPEG-2 Video and Systems syntax---including B-pictures. Both interlaced(1920 x 1080 x 30 Hz) and progressive (1280 x 720 x 60 Hz) modes will be supported. The Alliance has also settled upon a modulation method (VSB) convolution coding (Viterbi), and error correction (Reed-Soloman) specification.

In September 1993, the consortium of 85 European companies signed an agreement to fund a project known Digital Video Broadcasting (DVB) which will develop a standard for cable and terrestrial transmission by the end of 1994. The scheme will use MPEG-2. This consortium has put the final nail in the coffin of the D-MAC scheme for gradual migration towards an all-digital, HDTV consumer transmission standard. The only remaining analog or digital-analog hybrid system left in the world is NHK's MUSE (which will probably be axed in a few years as soon as it appears to be politically secure thing to do).

75. What is the best compression ratio for MPEG ?
A. The MPEG sweet spot is about 1.2 bits/pel Intra and .35 bits/pel inter. Experimentation has shown that intra frame coding with the familiar DCT-Quantization-Entropy hybrid algorithm achieves optimal performance at about an average of 1.2 bits/sample or about 6:1 compression ratio. Below this point, artifacts become noticeable.

76. Is there a MPEG CD-ROM format?
A. Yes, a consortium of international companies (Matsushita, Philips, Sony, JVC, et al) have agreed upon a specification for MPEG video and audio. 2 hour long movies are stored on two 650 MByte compact discs. The video
rate is 1.15 Mbit/sec, the audio rate is either 128 kbit/sec or 192 kbit/sec Layer I or Layer II.(this seems to contradict the Philips 224 kbit/s audio spec?). Although the Video, Systems, and Audio syntax are identical, the CD-I movie format and the White Book format are not compatible.

Researchers are busy experimenting with denser and faster rate CD formats, perhaps using green or blue laser wavelengths. One demonstration stretched the pit and track density to its limits, improving areal density by almost 2 fold.

MZ EXE Format
Intel byte order

Information from File Format List 2.0 by Max Maischein.

--------!-CONTACT_INFO----------------------
If you notice any mistakes or omissions, please let me know! It is only
with YOUR help that the list can continue to grow. Please send
all changes to me rather than distributing a modified version of the list.

This file has been authored in the style of the INTERxxy.* file list
by Ralf Brown, and uses almost the same format.

Please read the file FILEFMTS.1ST before asking me any questions. You may find
that they have already been addressed.

Max Maischein

Max Maischein, 2:244/1106.17
Max_Maischein@spam.fido.de
corion@informatik.uni-frankfurt.de
Corion on #coders@IRC
--------!-DISCLAIMER------------------------
DISCLAIMER: THIS MATERIAL IS PROVIDED "AS IS". I verify the information
contained in this list to the best of my ability, but I cannot be held
responsible for any problems caused by use or misuse of the information,
especially for those file formats foreign to the PC, like AMIGA or SUN file
formats. If an information it is marked "guesswork" or undocumented, you
should check it carefully to make sure your program will not break with
an unexpected value (and please let me know whether or not it works
the same way).

Information marked with "???" is known to be incomplete or guesswork.

Some file formats were not released by their creators, others are regarded
as proprietary, which means that if your programs deal with them, you might
be looking for trouble. I don't care about this.

The old EXE files are the EXE files executed directly by MS-DOS. They were a
major improvement over the old 64K COM files, since EXE files can span multiple
segments. An EXE file consists of three different parts, the header, the
relocation table and the binary code.
The header is expanded by a lot of programs to store their copyright information
in the executable, some extensions are documented below.
The format of the header is as follows :

```
OFFSET              Count TYPE   Description
0000h                   2 char   ID='MZ'
                                 ID='ZM'
0002h                   1 word   Number of bytes in last 512-byte page
                                 of executable
0004h                   1 word   Total number of 512-byte pages in executable
                                 (including the last page)
0006h                   1 word   Number of relocation entries
0008h                   1 word   Header size in paragraphs
000Ah                   1 word   Minimum paragraphs of memory allocated in
                                 addition to the code size
000Ch                   1 word   Maximum number of paragraphs allocated in
                                 addition to the code size
000Eh                   1 word   Initial SS relative to start of executable
0010h                   1 word   Initial SP
0012h                   1 word   Checksum (or 0) of executable
0014h                   1 dword  CS:IP relative to start of executable
                                 (entry point)
0018h                   1 word   Offset of relocation table;
                                 40h for new-(NE,LE,LX,W3,PE etc.) executable
001Ah                   1 word   Overlay number (0h = main program)
```

Following are the header expansions by some other prorams like TLink, LZExe and
other linkers, encryptors and compressors; all offsets are relative to the start
of the whole header :

---new executable

```
OFFSET              Count TYPE   Description
001Ch                   4 byte   ????
0020h                   1 word   Behaviour bits ??
0022h                  26 byte   reserved (0)
003Ch                   1 dword  Offset of new executable header from start of
                                 file (or 0 if plain MZ executable)

---Borland TLINK
OFFSET              Count TYPE   Description
001Ch                   2 byte   ?? (apparently always 01h 00h)
001Eh                   1 byte   ID=0FBh
001Fh                   1 byte   TLink version, major in high nybble
0020h                   2 byte   ??

---old ARJ self-extracting archive
OFFSET              Count TYPE   Description
001Ch                   4 char   ID='RJSX' (older versions)
                                 new signature is 'aRJsf'" in the first 1000
                                 bytes of the file)
---LZEXE compressed executable
OFFSET              Count TYPE   Description
001Ch                   2 char   ID='LZ'
001Eh                   2 char   Version number :
                                  '09' - LZExe 0.90
                                  '91' - LZExe 0.91
---PKLITE compressed executable
OFFSET              Count TYPE   Description
001Ch                   1 byte   Minor version number
001Dh                   1 byte   Bit mapped :
                                 0-3 - major version
                                   4 - Extra compression
                                   5 - Multi-segment file
001Eh                   6 char   ID='PKLITE'
---LHarc 1.x self-extracting archive
OFFSET              Count TYPE   Description
001Ch                   4 byte   unused???
0020h                   3 byte   Jump to start of extraction code
0023h                   2 byte   ???
0025h                  12 char   ID='LHarc's SFX '
--LHA 2.x self-extracting archive
OFFSET              Count TYPE   Description
001Ch                   8 byte   ???
0024h                  10 char   ID='LHa's SFX '
                                 For version 2.10
                                 ID='LHA's SFX ' (v2.13)
                                 For version 2.13
---LH self-extracting archive
OFFSET              Count TYPE   Description
001Ch                   8 byte   ???
0024h                   8 byte   ID='LH's SFX '
---TopSpeed C 3.0 CRUNCH compressed file
OFFSET              Count TYPE   Description
001Ch                   1 dword  ID=018A0001h
0020h                   1 word   ID=1565h
---PKARC 3.5 self-extracting archive
OFFSET              Count TYPE   Description
001Ch                   1 dword  ID=00020001h
0020h                   1 word   ID=0700h
---BSA (Soviet archiver) self-extracting archive
OFFSET              Count TYPE   Description
001Ch                   1 word   ID=000Fh
001Eh                   1 byte   ID=A7h
---LARC self-extracting archive
OFFSET              Count TYPE   Description
001Ch                   4 byte   ???
0020h                  11 byte   ID='SFX by LARC '

After the header, there follow the relocation items, which are used to span
multpile segments. The relocation items have the following format :
OFFSET              Count TYPE   Description
0000h                   1 word   Offset within segment
0002h                   1 word   Segment of relocation
To get the position of the relocation within the file, you have to compute the
```

physical adress from the segment:offset pair, which is done by multiplying the segment by 16 and adding the offset and then adding the offset of the binary start. Note that the raw binary code starts on a paragraph boundary within the executable file. All segments are relative to the start of the executable in memory, and this value must be added to every segment if relocation is done manually.

EXTENSION:EXE,OVR,OVL
OCCURENCES:PC
PROGRAMS:MS-DOS
REFERENCE:Ralf Brown's Interrupt List
SEE ALSO:COM,EXE,NE EXE

NCSA HDF
Specification and Developer's Guide

Version 3.2

September 1993

University of Illinois at Urbana-Champaign

NCSA HDF Version 3.3 source code and documentation are in the public domain. Specifically, we give to the public domain all rights for future licensing of the source code, all resale rights, and all publishing rights.

We ask, but do not require, that the following message be included in all derived works: *Portions developed at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign.*

READ ME NOW

If you want to see more software like NCSA HDF, you need to send us a letter, email or US mail, telling us what you are doing with NCSA HDF. We need to know: (1) What science you are working on—an abstract of your work would be fine; and (2) How NCSA HDF has helped you, for example, by increasing your productivity or allowing you to do things you could not do before.

We encourage you to cite the use of NCSA HDF, and any other NCSA software you have used, in your publications. A bibliography of your work would be extremely helpful.

NOTE: This is a new kind of shareware. You share your science and successes with us, and we can get more resources to share more software like NCSA HDF with you.

NCSA Contacts

Mail user feedback, bugs, and software and manual suggestions to:

NCSA Software Tools Group
HDF
152 Computing Applications Bldg.
605 E. Springfield Ave.
Champaign, IL 61820

Send commmunications via electronic mail to one of the following:

Bug Suggestions
bugs@ncsa.uiuc.edu
bugs@ncsavmsa.bitnet

All Other Communications
softdev@ncsa.uiuc.edu
softdev@ncsavmsa.bitnet

Disclaimer

THE UNIVERSITY OF ILLINOIS GIVES NO WARRANTY, EXPRESS OR IMPLIED, FOR THE SOFTWARE AND/OR DOCUMENTATION PROVIDED, INCLUDING, WITHOUT LIMITATION, WARRANTY OF MERCHANTIBILITY AND WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE.

Trademark Acknowledgments

Macintosh and Macintosh II are trademarks of Apple Computer Inc.
UNIX is a registered trademark of AT&T.

CRAY and UNICOS are registered trademarks and CRAY-2 and CFT77 are trademarks of Cray Research Inc.
IBM PC is a registered trademark of International Business Machines Corporation.
MS-DOS is a registered trademark of Microsoft Corporation.
Sun is a registered trademark and Sun Workstation and Sun System 3 are trademarks of Sun Microsystems Inc.

Table of contents

Introduction

Overview vi
Why HDF? vi
What is HDF? vii
Some History ix
About This Document x
Conventions Used in This Document xi

Chapter 1 Basic Structure of HDF Files

Chapter Overview 1-1
File Header 1-1
Data Objects 1-1
Physical Organization of HDF Files 1-4
Sample HDF File 1-5

Chapter 2 Software Overview

Chapter Overview 2-1
HDF Software Layers 2-1
Software Organization 2-2
Some HDF Conventions 2-6

Chapter 3 General Purpose Interface

Chapter Overview 3-1

Introduction 3-1

New Low Level Routines with Version 3.2 3-2

Overview of the Interface 3-2

Function Specifications 3-6

Chapter 4 Sets and Groups

Chapter Overview 4-1

Data Sets 4-1

Groups 4-2

Raster Image Sets (RIS) 4-5

Scientific Data Sets 4-8

Vsets, Vdatas, and Vgroups 4-14

The Raster-8 Set (Obsolete) 4-16

Chapter 5 Annotations

Chapter Overview 5-1

General Description 5-1

File Annotations 5-2

Object Annotations 5-2

Chapter 6 Tag Specifications

Chapter Overview 6-1

The HDF Tag Space 6-1

Extended Tags and Alternate Physical Storage Methods 6-1

Tag Specifications 6-7

Chapter 7 Portability Issues

Chapter Overview 7-1

The HDF Environment 7-1

Organization of Source Files 7-3

Passing Strings Between FORTRAN and C 7-5

Function Return Values between FORTRAN and C 7-7

Differences in Routine Names 7-8

Differences Between ANSI C and Old C 7-10

Type Differences 7-11

Access to Library Functions 7-14

Appendix A Tag and Extended Tag Table

Tags A-1

Extended Tag Labels A-4

I Introduction

Overview

The Hierarchical Data Format (HDF) was designed to be an easy, straight-forward, and self-describing means of sharing scientific data among people, projects, and types of computers. An extensible header and carefully crafted internal layers provide a system that can grow as scientific data-handling needs evolve.

This document, the *NCSA HDF Specification and Developer's Guide*, fully defines HDF and its interfaces, discusses criteria employed in its development, and provides guidelines for developers working on HDF itself or building applications that employ HDF.

This introduction provides a brief overview of HDF capabilities and design.

Why HDF?

A fundamental requirement of scientific data management is the ability to access as much information in as many ways, as quickly and easily as possible. A data storage and retrieval system that facilitates these capabilities must provide the following features:

Support for scientific data and metadata
Scientific data is characterized by a variety of data types and representations, data sets (including images) that can be extremely large and complex, and the need to attach accompanying attributes, parameters, notebooks, and other metadata. Metadata, supplementary data that describes the basic data, includes information such as the dimensions of an array, the number type of the elements of a record, or a color lookup table (LUT).

Support for a range of hardware platforms
Data can originate on one machine only to be used later on many different machines. Scientists must be able to access data and metadata on as many hardware platforms as possible

Support for a range of software tools
Scientists need a variety of software tools and utilities for easily searching, analyzing, archiving, and transporting the data and metadata. These tools range from a library of routines for reading and writing data and metadata, to small utilities that simply display an image on a console, to full-blown database retrieval systems that provide multiple views of thousands of sets of data and metadata.

Rapid data transfer
Both the size and the dispersion of scientific data sets require that mechanisms exist to get the data from place to place rapidly.

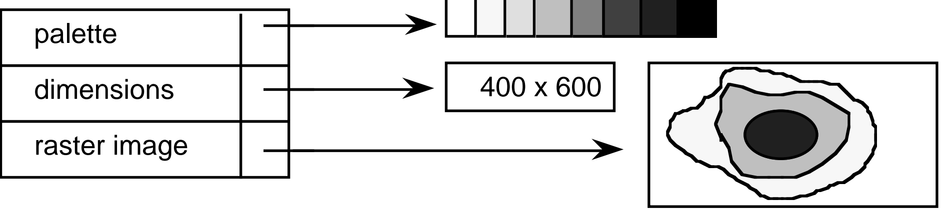
Extendibility
As new types of information are generated and new kinds of science are done, a means must be provided to support them.

What is HDF?

The HDF Structure

HDF is a self-describing extensible file format using tagged objects that have standard meanings. The idea is to store both a known format description and the data in the same file. HDF tags describe the format of the data because each tag is assigned a specific meaning: the tag `DFTAG_LUT` stands for color palette, the tag `DFTAG_RI` stands for 8bit raster image, and so on (see Figure 1). A program that has been written to understand a certain set of tag types can scan the file for those tags and process the data. This program also can ignore any data that is beyond its scope.

Figure I.1 Raster Image Set in an HDF File . The set has three data objects with different tags representing three different types of data. The palette and dimension objects contain metadata.



The set of available data objects encompasses both primary data and metadata. Most HDF objects are machine- and medium-independent, physical representations of data and metadata.

HDF Tags

The HDF design assumes that we cannot know *a priori* what types of data objects will be needed in the future, nor can we know how scientists will want to view that data. As science progresses, people will discover new types of information and new relationships among existing data. New types of data objects new tags will be created to meet these expanding needs. To avoid unnecessary proliferation of tags and to ensure that all tags are available to potential users who need to share data, a portable public domain library is available that interprets all public tags. The library contains user interfaces designed to provide views of the data that are most natural for users. As we learn more about the way scientists need to view their data, we can add user interfaces that reflect data models consistent with those views.

Types of Data and Structures

HDF currently supports the most common types of data and metadata that scientists use, including multidimensional gridded data, 2-dimensional raster images, polygonal mesh data, multivariate data sets, finite-element data, non-Cartesian coordinate data, and text.

In the future there will almost certainly be a need to incorporate new types of data, such as voice and video, some of which might actually be stored on other media than the central file itself. Under such circumstances, it may become desirable to employ the concept of a *virtual file*. A virtual file functions like a regular file but does not fit our normal notion of a monolithic sequence of bits stored entirely on a single disk or tape.

HDF also makes it possible for the user to include annotations, titles, and specific descriptions of the data in the file. Thus, files can be archived with human-readable information about the data and its origins

One collection of HDF tags supports a hierarchical grouping structure called *Vset* that allows scientists to organize data objects within HDF files to fit their views of how the objects go together, much as a person in an office or laboratory organizes information in folders, drawers, journal boxes, and on their desktops.

Backward and Forward Compatibility

An important goal of HDF is to maximize backward and forward compatibility among its interfaces. This is not always achievable, because data formats must sometimes change to enhance performance, to correct errors, or for other reasons. However, whenever possible, HDF files should not become out of date. For example, suppose a site falls far behind in the HDF standard so its users can only work with the portions of the specification that are three years old. Users at this site might produce files with their old HDF software then read them with newer software designed to work with more advanced data files. The newer software should still be able to read the old files.

Conversely, if the site receives files that contain objects that its HDF software does not understand, it should still be able to list the types of data in the file. It should also be able to access all of the older types of data objects that it understands, despite the fact that the older types of data objects are mixed in with new kinds of data. In addition, if the more advanced site uses the text annotation facilities of HDF effectively, the files will arrive with complete human-readable descriptions of how to decipher the new tag types.

Calling Interfaces

To present a convenient user interface made up of something more usable than a list of tag types with their associated data requirements, HDF supports multiple calling interfaces.

The *low level calling interfaces* are used to manipulate tags and raw data, for error handling, and to control the physical storage of data. These interfaces are designed to be used by developers who are providing the higher level interfaces for applications like raster image storage or scientific data archiving.

The *application interfaces*, at the next level, include several modules specifically designed to simplify the process of storing and accessing

specific types of data. For example, the palette interface is designed to handle color palettes and lookup tables while the scientific data interface is designed to handle arrays of scientific data. If you are primarily interested in reading or writing data to HDF files, you will spend most of your time working with the application interfaces.

The *HDF utilities* and *NCSA applications*, at the top level, are special purpose programs designed to handle specific tasks or solve specific problems. The utilities provide a command line interface for data management. The applications provide solutions for problems in specific application areas and often include a graphic user interface. Several *third party applications* are also available at this level.

Machine Independence

An important issue in data file design is that of machine independence or transportability. The HDF design defines standard representations for storing all data types that it supports. When data is written to a file, it is typically written in the standard HDF representation. The conversion is handled by the HDF software and need not concern the user. Users may override this convention and install their own conversion routines, or they may write data to a file in the native format of the machine on which it was generated.

Some History

In 1987 a group of users and software developers at NCSA searched for a file format that would satisfy NCSA's data needs. There were some interesting candidates, but none that were in the public domain, were targeted to scientific data, and yet were sufficiently general and extensible. In the course of several months, borrowing concepts from several existing formats, the group designed HDF.

The first version of HDF was implemented in the spring and summer of 1988. It included a general purpose interface and an 8-bit raster image interface. In the fall of 1988, a scientific data set interface was designed and implemented, enabling HDF users to store multidimensional arrays and related data. Soon thereafter interfaces were implemented for storing color palettes, 24-bit raster images, and annotations.

In 1989, it became clear that there was a need to support a general grouping structure and unstructured data such as that used to represent polyhedra in graphical applications. This led to Vsets, whose interface routines were implemented as a separate HDF library.

Also in 1989 it became clear that the existing general purpose layer was not sufficiently powerful to meet anticipated future needs and that the coding could use a substantial overhaul. From this, the long process of redesigning the lower layers of HDF began. The first version incorporating extended tags and the new lower layers of HDF was released in the summer of 1992 as HDF Version 3.2.

This release, HDF Version 3.3, provides alternative physical storage methods (external and linked block data elements) through extended tags, JPEG data compression, changes to some Vset interface functions,

access to netCDF files through a complete netCDF interface,[1] hyperslab access routines for old-style SDS objects, and various performance improvements.

About This Document

This document is designed for software developers who are designing applications or routines for use with HDF files and for users who need detailed information about HDF. Users who are interested in using HDF to store or manipulate their data will not normally need the kind of detail presented in this manual. They should instead consult one of the user-level documents:

- Versions 3.2 and earlier
  - *NCSA HDF Calling Interfaces and Utilities*
  - *NCSA HDF Vset*
- Version 3.3
  - *Getting Started with NCSA HDF*
  - *NCSA HDF User's Guide*
  - *NCSA HDF Reference Manual*

Someone using third-party software that uses HDF may also have to consult a manual for that software.

Document Contents

The *NCSA HDF Specification and Developer's Guide* contains the following chapters and appendix:

Chapter 1: Basic Structure of HDF Files
Introduces and describes the components and organization of HDF files

Chapter 2: Software Overview
Describes the organization of the software layers that make up the basic HDF library and provides guidelines for writing HDF software

Chapter 3: General Purpose Interface
Describes the low level HDF routines that make up the general purpose interface

Chapter 4: Sets and Groups
Explains the roles of sets and groups in an HDF file, and describes raster image sets, scientific data sets, and Vsets

Chapter 5: Annotations
Explains the use of annotations in HDF files

Chapter 6: Tag Specifications
Describes the tag identification space, the extended tag structure, and all of the NCSA-supported tags

Chapter 7: Portability Issues
Describes the measures taken to maximize HDF portability across platforms and to ensure that HDF routines are available to both C and FORTRAN programs

1 NetCDF is a network-transparent derivative of the original CDF (Common Data Format) developed by the National Aeronautics and Space Administration (NASA). It is used widely in atmospheric sciences and other disciplines requiring very large data structures. NetCDF is in the public domain and was developed at the Unidata Program Center in Boulder, Colorado.

Appendix A: Tags and Extended Tag Labels
Presents a list of NCSA-supported HDF tags and a list of labels used with extended tags

Conventions Used in This Document

Most of the descriptive text in this guide is printed in 10 point New Century Schoolbook. Other typefaces have specific meanings that will help the reader understand the functionality being described.

New concepts are sometimes presented in italics on their first occurrence to indicate that they are defined within the paragraph.

Cross references within the specification include the title of the referenced section or chapter enclosed in quotation marks. (E.g., See Chapter 1, "The Basic Structure of HDF Files," for a description of the basic HDF file structure.)

References to documents italicize the title of the document. (E.g., See the guide *Getting Started with NCSA HDF* to familiarize yourself with the basic principles of using HDF.)

Literal expressions and *variables* often appear in the discussion. Literal expressions are presented in Courier while variables are presented in italic Courier. A literal expression is any expression that would be entered exactly as presented, e.g., commands, command options, literal strings, and data. A variable is an expression that serves as a place holder for some other text that would be entered. Consider the expression `cp` *`file1 file2`*. `cp` is a command name and would be entered exactly as it appears, so it is printed in bold Courier. But *`file1`* and *`file2`* are variables, place holders for the names of real files, so they are printed in italic bold Courier; the user would enter the actual filenames.

This guide frequently offers sample *command lines*. Sometimes these are examples of what might be done; other times they are specific instructions to the user. Command lines may appear within running text, as in the preceding paragraph, or on a separate line, as follows:

```
cp file1 file2
```

Command lines always include one or more literal expressions and may include one or more variables, so they are printed in Courier and italic Courier as described above.

Keys that are labeled with more than one character, such as the RETURN key, are identified with all uppercase letters. Keys that are to be pressed simultaneously or in succession are linked with a hyphen. For example, “press CONTROL-A” means to press the CONTROL key then, without releasing the CONTROL key, press the A key. Similarly, “press CONTROL-SHIFT-A “ means to press the CONTROL and SHIFT keys then, without releasing either of those, press the A key.

Table I.1 summarizes the use of typefaces in the technical discussion (i.e., everything except references and cross references).

Table I.1 Meaning of entry format notations

Type	Appearance	Example	Entry Method
Literal expression (commands, literal strings, data)	Courier	`dothis`	Enter the expression exactly as it appears.
Variables	Italic Courier	*`filename`*	Enter the name of the file or the specific data that this expression represents.
Special keys	Uppercase	RETURN	Press the key indicated.
Key combinations	Uppercase with hyphens between key names	CONTROL-A	While holding down the first one or two keys, press the last key.

Program listings and *screen listings* are presented in a boxed display in Courier type such as in Figure I.2, "Sample Screen Listing." When the listing is intended as a sample that the reader will use for an exercise or model, variables that the reader will change are printed in italic Courier.

Figure I.2 Sample screen listing

```
mars_53% ls -F
MinMaxer/                      net.source
mars_54% cd MinMaxer
mars_55% ls -F
list.MinMaxer                  minmaxer.v1.04/
mars_56% cd minmaxer.v1.04
mars_57% ls -F
COPYRIGHT                      minmaxer.bin/                source.minmaxer/
README                         sample/                      source.triangulation/
mars_58%
```

Chapter 1 Basic Structure of HDF Files

Chapter Overview

This chapter introduces and describes the components and organization of Hierarchical Data Format (HDF) files.

File Header

The first component of an HDF file is the *file header* (FH), which takes up the first four bytes in an HDF file. The file header is a signature that indicates that the file is an HDF file. Specifically, it is a 32-bit magic number with the hexadecimal value 0e031301.

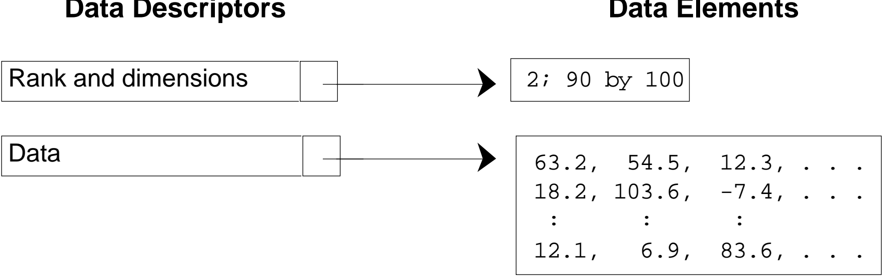
Note: To ensure portability, the programmer must ensure that the hexadecimal value in an HDF file header is written in big-endian order.

HDF assumes big-endian order in reading and writing files. The order of bytes in the file header might be swapped on some machines when the HDF file header is written, causing these characters to be written in little-endian order. To maintain HDF file portability when developing software for such machines, you must make sure the characters are read and written in the exact order shown.

Data Objects

The basic building block of an HDF file is the *data object*, which contains both data and information about the data. A data object has two parts: a 12-byte *data descriptor* (DD) and a *data element*. Figure 1.1 illustrates two data objects.

Figure 1.1 Two Data Objects



As the names imply, the data descriptor provides information about the data; the data element is the data itself. In other words, all data in an

HDF file has information about itself attached to it. In this sense, HDF files are *self-describing* files.

Data Descriptor (DD)

A data descriptor (DD) has four fields: a 16-bit tag, a 16-bit reference number, a 32-bit data offset, and a 32-bit data length. These are depicted in Figure 1.2 and are briefly described in Table 1.1. Explanations of each part appear in the paragraphs following Table 1.1.

Figure 1.2 A Data Descriptor (DD)

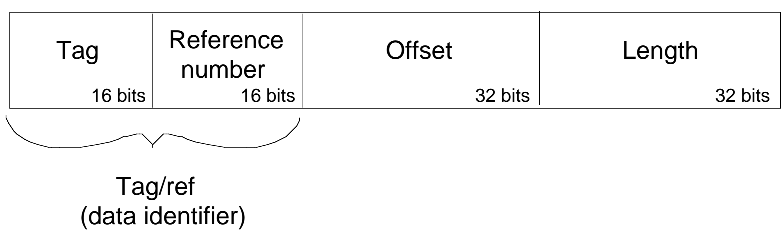


Table 1.1 Parts of a Data Descriptor

Part	Description	
Tag/ref (data identifier)	Unique identifier for each data element	
	Tag	Type of data in a data element
	Reference number	Number distinguishing data element from others with the same tag
Offset	Byte offset of data element from beginning of file	
Length	Length of data element	

Note: Only the full tag/ref uniquely identifies a data element.

Tag/ref (Data Identifier)

A tag and its associated reference number (abbreviated as tag/ref) uniquely identify a data element in an HDF file. The tag/ref combination is also known as a *data identifier*.

Tag

A *tag* is the part of a data descriptor that tells what kind of data is contained in the corresponding data element. A tag is actually a 16-bit unsigned integer between 1 and 65535, but every tag is also given a name that programs can refer to instead of the number. If a DD has no corresponding data element, its tag is `DFTAG_NULL`, indicating that no data is present. A tag may never be zero.

Tags are assigned by NCSA as part of the specification of HDF. The following ranges are to be used to guide tag assignment:

00001 – 32767 reserved for NCSA use
32768 – 64999 user-definable
65000 – 65535 reserved for expansion of the format

Chapter 6, "Tag Specifications," provides full specifications for all currently supported HDF tags. Appendix A, "Tags and Extended Tag Labels," lists the current tag assignments. See the section "Some HDF Conventions" in Chapter 2, "Software Overview," for more information on allocating tags.

Reference Number

Tags are not necessarily unique in an HDF file; there may be more than one data element of a given type. Therefore, each tag is associated with a unique *reference number* in the data descriptor.

Reference numbers are not necessarily assigned consecutively, so you cannot assume that the actual value of a reference number has any meaning beyond providing a way of distinguishing among elements with the same tag. Furthermore, reference numbers are only unique for data elements with the same tag; two 8-bit raster images will never have the same reference number but an 8-bit raster image and a 24-bit raster image might.

Reference numbers are 16-bit unsigned integers.

Data Offset and Length

Note: All offsets are from the beginning of the file; they are not relative.

The *data offset* states the byte position of the corresponding data element from the beginning of the file. The *length* states the number of bytes occupied by the data element.

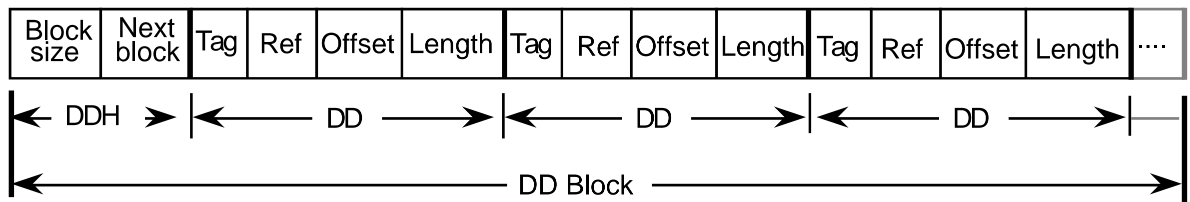
Offset and length are both 32-bit unsigned integers.

DD Blocks

Data descriptors are stored physically in a linked list of blocks called *data descriptor blocks* or DD blocks. The individual components of a DD block are depicted in Figure 1.3. All of the DDs in a DD block are assumed to contain significant data unless they have the tag DFTAG_NULL (no data).

In addition to its DDs, each data descriptor block has a *data descriptor header* (DDH). The DDH has two fields: a *block size* field and a *next block* field. The block size field is a 16-bit unsigned integer that indicates the number of DDs in the DD block. The next block field is a 32-bit unsigned integer giving the offset of the next DD block, if there is one. The DDH of the last DD block in the list contains a 0 in its next block field.

Figure 1.3 Model of a Data Descriptor Block



Since the default number of DDs in a DD block is defined when the HDF library is compiled, changing the default requires recompilation.

Data Element

A *data element* is the raw data portion of a data object. Its data type can be determined by examining its tag, but other interpretive information may be required before it can be processed properly.

Each data element is stored as a set of contiguous bytes starting at the offset and with the length specified in the corresponding DD.[1]

Exceptions

Note that the data object identified by the tag `DFTAG_MT` does not adhere to the standards described above; it consists of the tag immediately followed by four number types. Since there can be only one `DFTAG_MT` tag in an HDF file, there is no need for a reference number. Since all the data can be stored in the DD with the tag, there is no need for a data element and the offset and length are unnecessary.

Several other tags, such as `DFTAG_NULL` and `DFTAG_JPEG`, serve as binary flags and convey all the required information by the mere fact of their presence in an HDF file. These tags therefore point to no data element and have offset and length values of `0`. Consider these examples: `DFTAG_NULL` indicates a data object containing no data; `DFTAG_JPEG` indicates that an associated data object, indicated by another tag, contains a JPEG data image. The descriptions of these tags include a *sink pointer* ( )in the diagrams in Chapter 6.

See the related entries in Chapter 6, "Tag Specifications," for a complete descriptions of these tags.

Physical Organization of HDF Files

The file header, DD blocks, and data elements appear in the following order in an HDF file:

- File header
- First DD block
- Data elements
- If necessary, more DD blocks, more data elements, etc.

These relationships are summarized in Table 1.2.

The only rule governing the distribution of DD blocks and data elements within a file is that the first DD block must follow immediately after the file header. After that, the pointers in the DD headers connect the DD blocks in a linked list and the offsets in the individual DDs connect the DDs to the data elements.

Table 1.2 Summary of the Relationships among Parts of an HDF File

Part	Constituents
HDF file	FH, DD block, data, DD block, data, DD block, data...
FH	0x0e031301 [32-bit HDF magic number]
DD block	DDH, DD, DD, DD, ...
DDH	Number of DDs [16 bits], offset to next DD block [32 bits]
DD	Tag [16 bits], ref [16 bits], offset [32 bits], length [32 bits]
Data	Data element, data element, data element ...

FH = file header, DD = data descriptor, DDH = DD header

[1] Some HDF software provides the capability of storing objects as a series of linked blocks or external elements, but this occurs at a higher level. At the lowest level each object with a tag/ref is stored contiguously.

Sample HDF File

We are now ready to examine a sample file. Consider an HDF file that contains two 400-by-600 8-bit raster images as described in Table 1.3.

Table 1.3 Sample Data Objects in an HDF File

Tag	Ref	Data
DFTAG_FID	1	File identifier: user-assigned title for file
DFTAG_FD	1	File descriptor: user-assigned block of text describing overall file contents
DFTAG_LUT	1	Image palette (768 bytes)
DFTAG_ID	1	*x*- and *y*-dimensions of the 2-dimensional arrays that contain the raster images (4 bytes)
DFTAG_RI	1	First 2-dimensional array of raster image pixel data (*x**y* bytes)
DFTAG_RI	2	Second 2-dimensional array of pixel data (also *x**y* bytes)

Assuming that a DD block contains 10 DDs, the physical organization of the file could be described by Figure 1.5.

In this instance, the file contains two raster images. The images have the same dimensions and are to be used with the same palette, so the same data objects for the palette (DFTAG_IP8) and dimension record (DFTAG_ID8) can be used with both images.

Figure 1.5 Physical Representation of Data Objects

Section	Item	Offset	Contents
Header	FH	0	0e031301 *(HDF magic number, in hexadecimal)*
DD block	DDH	4	10 0
	DD	10	DFTAG_FID 1 130 4
	DD	22	DFTAG_FD 1 134 41
	DD	34	DFTAG_LUT 1 175 768
	DD	46	DFTAG_ID 1 943 4
	DD	58	DFTAG_RI 1 947 240000
	DD	70	DFTAG_RI 2 240947 240000
	DD	82	DFTAG_NULL *(Empty)*
	DD	94	DFTAG_NULL *(Empty)*
	DD	106	DFTAG_NULL *(Empty)*
	DD	118	DFTAG_NULL *(Empty)*
Data	Data	130	sw3
	Data	134	solar wind simulation: third try. 8/8/88
	Data	175	.... *(Data for the image palette)*
	Data	943	400 600 *(Image dimensions)*
	Data	947	.... *(Data for the first raster image)*
	Data	240947	.... *(Data for the second raster image)*

Chapter 2 Software Overview

Chapter Overview

This chapter describes the HDF software organization and provides guidelines for writing HDF software.

HDF is an amalgam of code and functionality from many sources. For example, the netCDF code came from the Unidata Program Center, and data compression and conversion software has been acquired from a variety of third parties. NCSA staff wrote the code for the basic HDF functionality and perfomed all of the integration work.

This document contains specifications for the NCSA-developed code and functionality. It does not include specifications for code or functionality from non-NCSA sources, though it does sometimes refer to specifications provided by other sources. Only the HDF interface to such code is specified in this document.

HDF Software Layers

There are three basic levels of HDF software:

- The HDF low level interface
- The HDF application interfaces
- HDF applications and utilities

The lowest layer, the *low level interface*, includes general purpose routines that form the basis of all higher-level HDF development. The low level routines directly execute functions such as file I/O, error handling, memory management, and physical storage.

The *application interfaces* support higher level views of data and provide the interfaces for building user-level applications. Routines to handle raster images, palettes, annotations, scientific data sets, Vdatas and netCDF appear at this level.

The *applications and utilities* are implemented at the highest level. NCSA utilities, NCSA applications, and third party applications are all implemented at this level.

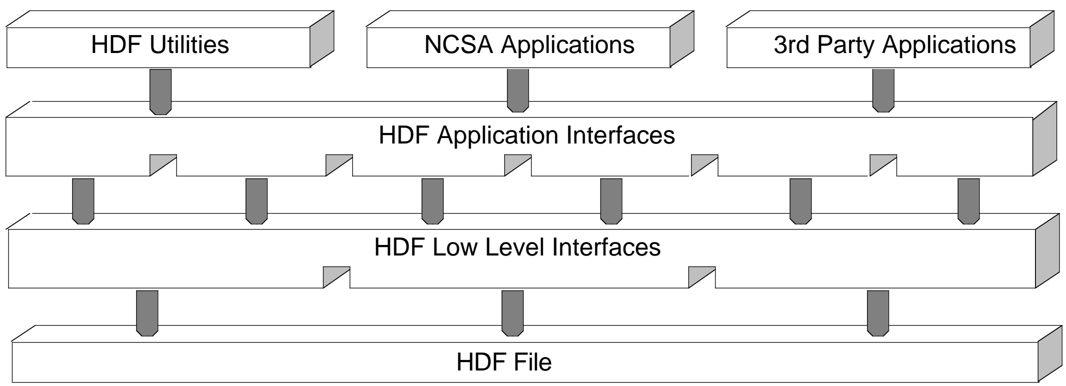
The utilities perform general functions, such as listing the contents of an HDF file, and more specialized functions, such as converting data from one HDF data type to another (e.g., raster images to scientific data sets). In general, the utilities have simple command line interfaces and perform data management tasks.

The applications usually perform data analysis tasks and have polished interactive user interfaces. They include the NCSA Visualization Tool

Suite, commercial software packages that use HDF, and other packages created at NCSA and by various third party projects.

Figure 2.1 illustrates this layered implementation.

Figure 2.1. HDF Software Layers [1]



The general purpose interfaces are described in detail in this document. The application interfaces and command line utilities are described in the document *NCSA HDF Calling Interfaces and Utilities* for Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* for Version 3.3. Other HDF-based software tools should have their own manuals.

Since the NCSA user community writes programs primarily in C and FORTRAN, all of the HDF application interfaces developed at NCSA are callable from both C and FORTRAN programs. Since the general purpose interface is primarily for program development, not for applications, it provides C-callable routines only.

Software Organization

Versions and Release Numbers

Since HDF is under continual development, new releases are periodically made available. Each new release of the HDF library is identified by a *version number*.

The version number consists of three elements:

`majorv` Major version number
`minorv` Minor version number
`rn` Release number

The version number is presented in the following format:

`majorv.minorvrrn` (e.g., Version 3.2r1)

These elements are interpreted as follows:

Major version number
A new major version number is assigned when there is some fundamental difference between a new version of the library and the

1 This is a simplified illustration of the HDF software layers. Though the basic principles illustrated here continue to apply, the introduction of netCDF and multiple-file HDF data structures renders the implementation considerably more complex.

previous version. When a new major version is released, HDF users and developers are strongly encouraged to obtain the new source code and documentation. There will probably be added functionality in successive major versions of the library and some obsolete code may be deleted. Some user code may have to be modified to use the new library.

Minor version number
A new minor version number indicates an intermediate release between one major version and the next. Changes will probably be significant. When a new minor version is released, users and developers are strongly encouraged to obtain the new source code and documentation.

Release number
A new release number is assigned when bug fixes or other small modifications have been made. Using a new release of the same version of the library will not usually require modifying existing user code.

ANSI C and Portability

To ensure that HDF can be easily ported to new platforms, all versions of the HDF source code from Version 3.2 on will be written in ANSI standard C, with special provisions for non-ANSI compilers. For more information about porting HDF and writing portable HDF-based code, refer to Chapter 7, "Making HDF Portable."

Modules and Interfaces

The HDF distribution contains many source files or modules that can be grouped into families. For example, `dfp.c`, `dfpf.c`, and `dfpff.f` all share the root name `dfp` and, therefore, all belong to the `dfp` family. In general, each family of source modules represents one HDF applications interface; the `dfp` family represents the HDF Palette Interface. Exceptions to this rule will be discussed later in this section.

For each interface, there is necessarily one file that contains the C code that provides the basic functionality of that interface. But some interfaces may have one or two additional code modules that provide FORTRAN callability for the interface, so families may have one, two, or three files:

1 file — Modules of this sort are generally not calling interfaces themselves; they provide useful support functions for actual calling interfaces. Since they are not meant to be called by any routine outside the HDF library, they do not need to be FORTRAN-callable. Example: `hblocks.c` is called only by internal HDF routines and has only the C-callable interface.

2 files — Although there are currently no two-file families, it is conceivable (and desirable) that some future interface will need only one extra source module to provide FORTRAN compatibility. If this were to happen, there would only be two source modules for the interface. Example: `dfnew.c` and `dfnewf.c` would make up the New Interface.

3 files — Most current implementations of FORTRAN-callable HDF interfaces require that character string arguments be passed to some of their functions. Due to differences in the way C and

FORTRAN represent strings, passing strings requires that there be a small amount of special purpose FORTRAN code written for each function that takes a string argument.

Therefore, most FORTRAN-callable HDF interfaces consist of three source modules:

- The primary C module
- A FORTRAN-callable C module
- A FORTRAN module

Example: `dfsd.c`, `dfsdf.c`, and `dfsdff.f` make up the Scientific Data Set Interface. `dfsd.c` contains the basic functionality of the interface. `dfsdf.c` provides the major part of FORTRAN callability. And `dfsdff.f` contains the special purpose FORTRAN code that enables passing character string arguments.

Header Files

In addition to the source code modules discussed above, some interfaces also have C header files associated with them that are meant to be included by C applications programmers with the `#include` preprocessor directive. They contain useful constants and data structures for interaction with the interface from C programs. The header files can be identified by the same name as the root name for the rest of the family with the `.h` extension. For example, `dfsd.h` is the header file for the Scientific Data Set Interface.

Of particular importance among the C header files are `hdf.h` and `hdfi.h`:

`hdf.h` Contains all the symbolic constants and public data structures required by HDF. `hdf.h` should be included by any program that uses any of these constants or data structures.

`hdfi.h` Contains specific portability information about each platform on which HDF is supported. `hdfi.h is` automatically included in programs when `hdf.h` is included, so programmers need not explicitly include it.

Refer to Chapter 7, "Making HDF Portable," for more information on `hdfi.h` and other portability issues.

By way of illustration, Table 2.1 lists selected families of source code modules and header files from of HDF Version 3.3.

Table 2.1 Sample HDF Version 3.3 Source Code Modules

General headers	General purpose	Grouping (non-Vset)	Utilities	Annota-tions	General rasters	Scientific data sets	Vsets
hdf.h	hfile.c	dfgroup.c	dfutil.c	dfan.c	dfgr.c	dfsd.c	vg.c
hdfi.h	hfilef.c	dfgroup.h	dfutilf.c	dfanf.c	dfgr.h	dfsdf.c	vgf.c
hproto.h	hfileff.f		dfutilff.f	dfanff.f	dfcomp.c	dfsdff.f	vgff.f
dfivms.h	hkit.c		dfutil.h	dfan.h	dfimcomp.c	dfsd.h	vfp.c
	hblocks.c				dfrig.h		vgi.h
	hextelt.c						vio.c
	herr.c						vconv.c
	herrf.c						vparse.c
	hfile.h						vrw.c
	herr.h						vsfld.c
							vg.h
							vproto.h

The HDF Test Suite

In addition to the source code for the HDF library, versions 3.2 and higher include a test suite. There are two test modules: one for C and one for FORTRAN. Each module tests all of the routines in all of the application interfaces and in the general purpose interface. The exact form of these test modules may vary from one release to the next; consult the release code and online test documentation for details.

Every effort has been made to ensure that the test programs provide a thorough and accurate assessment of the health of the HDF library. Although the test suite will greatly improve the reliability of HDF code, it is almost inevitable that some parts of the code will remain untested. Therefore, no guarantees can be made on the basis of test suite performance.

Sample HDF Programs

Each HDF release includes several sample programs to help users write HDF programs. They illustrate some of the common techniques employed by HDF programmers.

Some HDF Conventions

The HDF specification described in the previous chapter is not sufficient to guarantee its success. It is also important that HDF programmers and users adhere to certain conventions. Some guidelines are implicit in the discussions in other sections of this document. Others are presented in the document *NCSA HDF Calling Interfaces and Utilities* (for Versions 3.2 and earlier) or in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* (for Version 3.3).

Guidelines not covered elsewhere are introduced in this section.

Naming and Assigning Tags

Tags that are to be made available to a general population of HDF users should be assigned and controlled by NCSA. Tags of this type are given numbers in the range 1 to 32,767. If you have an application that fits this criterion, contact NCSA at the address listed in the front matter at the beginning of this manual and specify the tags you would like. For each tag, your specifications should include a suggested name, information about the type and structure of the data that the tag will refer to, and information about how the tag will be used. Your specifications should be similar to those contained in Chapter 6, "Tag Specifications." NCSA will assign a set of tags for your application and will include your tag descriptions in the HDF documentation.

Tags in the range 32,768 to 64,999 are user-definable. That is, you can assign them for any private application. If you use tags in this range, be aware that they may conflict with other people's private tags.

Using Reference Numbers to Organize Data Objects

Note: Users are discouraged from assigning any meaning to reference numbers beyond that imparted by the HDF library.

The HDF library itself uses reference numbers solely to distinguish among objects with the same tag. While application programmers may find it convenient to impart some meaning to reference numbers, they should be forewarned that the HDF library will be ignorant of any such meaning.

Multiple References

Multiple references to a single data element are quite common in HDF. The general purpose routine `Hdupdd` generates a new reference to data that is already pointed to by another DD. If `Hdupdd` is used several times, there may be several DDs that point to the same data element.

It is important to note that when a multiply-referenced data element is deleted or moved, the various DDs that previously pointed to the data element are *not* automatically deleted or adjusted to point to the data element in its new location. Consequently, each DD to be deleted or moved should be checked for multiple references and handled appropriately.

Chapter 3 General Purpose Interface

Chapter Overview

This chapter provides a detailed description of the routines that make up the HDF general purpose interface.

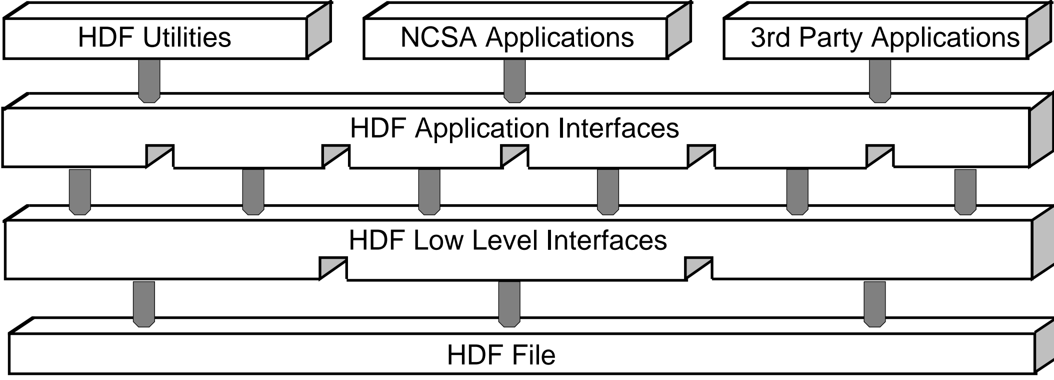
Introduction

HDF supports several interfaces which can be categorized as high level and general purpose interfaces:

- High level interfaces support utilities and applications.
- General purpose interfaces perform basic operations on HDF files.

These levels are illustrated in Figure 3.1, "HDF Software Layers."

Figure 3.1. HDF Software Layers



This chapter is concerned only with the general purpose routines.

Using these routines, you will be able to build and manipulate HDF objects of any type, including those of your own design. All HDF applications developed at NCSA use them as basic building blocks.

The general purpose routines are all written in C but are typically accessible from FORTRAN.

New General purpose Routines with Version 3.2

The general purpose routines described in this chapter were new with HDF Version 3.2, released in June 1992; they replace the routines provided with earlier versions. The new routines provide better

performance and increased functionality and users are strongly advised to use them in new applications. The old routines are supported through emulation, but may be eliminated from the HDF library in a future release.

The new lower layer incorporates the following improvements:

- More consistent data and function types
- More meaningful and extensive error reporting
- Simplification of key lower level functions
- Simplified techniques to facilitate portability
- Support for alternate forms of physical storage, such as linked blocks storage and storage of the data portion of an object in an external file
- A version tag to indicate which version of the HDF library last changed a file
- Support for simultaneous access to multiple files
- Support for simultaneous access to multiple objects within a single file

The previous lower layer was called the *DF layer* because all routines began with the letters `DF` (e.g., `DFopen` and `DFclose`). The new lower layer is called the *H layer* because all routines begin with the letter `H` (e.g., `Hopen`, `Hclose`, and `Hwrite`). The source modules containing these routines begin with the letter `h` (see Table 2.1, "HDF Version 3.2 source code modules"):

`hfile.c`	Basic I/0 routines
`herr.c`	Error-handling routines
`hkit.c`	General purpose routines
`hblocks.c`	Routines to support linked block storage
`hextelt.c`	Routines to support external storage of HDF data elements

Overview of the Interface

This section provides specifications and descriptions of the public functions of the general purpose interface.

Opening and Closing HDF Files

These calls are used to open and close HDF files:

`Hopen`	Provides an access path to an HDF file and reads all of the DD blocks in the file into memory
`Hclose`	Closes the access path to a file

Locating Elements for Access and Getting Information

These routines locate elements or acquire other information about an HDF file or its data objects. Except for `Hendaccess`, they initialize the element that they locate and return an *access ID* that is used in later references to the data element. Calls can include wildcards so that one can search for unknown tags and reference numbers (tag/refs).

`Hstartread`	Locates an existing data element with matching tag/ref and returns an access ID for reading it
`Hnextread`	Continues the search with the same access ID
`Hendaccess`	Disposes of access ID for tag/ref
`Hinquire`	Returns access information about a data element

`Hishdf`	Determines whether a file is an HDF file.
`Hnumber`	Returns the number of occurrences of a specified tag/ref in a file
`Hgetlibversion`	Returns version information for the current HDF library
`Hgetfileversion`	Returns version information for an HDF file

Reading and Writing Entire Data Elements

There are two sets of routines for reading and writing data elements. The routines described here are used to store and retrieve entire data elements.

`Hputelement`	Adds or replaces elements in a file
`Hgetelement`	Reads data elements in a file

A second set of routines, described in the next section, may be used if you wish to access only part of a data element.

Reading and Writing Part of a Data Element

The second set of routines for reading and writing data elements makes it possible to read or write all or part of a data element. One of the access routines `Hstartread` or `Hstartwrite` must be called before these `Hwrite`, `Hread`, or `Hseek`:

`Hstartwrite`
: Sets up writing to the object with the supplied tag/ref. If the object exists, it will be modified; otherwise it will be created.

`Hwrite`
: Writes data to a data element where the last write or Hseek() stopped. If the space reserved is less than the length to write, then only as much as can fit is written.

`Hread`
: Reads a portion of a data element. It starts at the last position left by an `Hread` or `Hseek` call and reads any data that remains in the element up to a specified number of bytes.

`Hseek`
: Sets the access pointer to an offset within a data element. The next time `Hread` or `Hwrite` is called, the access occurs from the new position. The location to seek can be specified as an offset from the current location, from the start of the element, or from the end of the element..

Manipulating Data Descriptors (DDs)

These routines perform operations on DDs without doing anything with the data to which the DDs refer:

`Hdupdd`	Generates new references to data that is already referenced from somewhere else
`Hdeldd`	Deletes a tag/ref from the list of DDs
`Hnewref`	Returns the next available reference number for the HDF file

Creating Special Data Elements

HDF 3.2 introduces two alternate methods of storing HDF objects: *linked blocks* and *external elements*. In previous releases, any data element had to be stored contiguously and all of the objects in an HDF file had to be in the same physical file. The contiguous requirement caused many problems, especially with regard to appending to existing objects. If you wanted to append data to an object, the entire data element had to be deleted and rewritten to the end of the file.

Linked blocks allow elements in a single HDF file to be non-contiguous.

External elements allow a single HDF object to be stored in an external file.

It is not currently possible to store a single object (such as a very large data set) in multiple files. Nor can multiple objects be stored in one external file.

Once they are created with the following routines, these special data elements can be accessed with the routines used for normal data elements:

`HLcreate`	Creates a new linked block special data element
`HXcreate`	Creates a new external file special data element

These routines have two modes of operation. Calling `HLcreate` with a tag/ref that does not exist in a file will create a new element with the given tag/ref which will be stored as linked blocks. On the other hand, if the tag/ref already exists in the file, the referenced object will be promoted to linked block status. All data which had been stored in the object before the promotion will be retained. `HXcreate` behaves similarly.

Development Routines

The HDF library provides the following developer-level routines that simplify the task of writing HDF applications. Most of these routines mirror basic C library functions which are, unfortunately, not always completely portable in their library form:

`HDgettagname`	Returns a pointer to a text string describing a given tag
`HDgetspace`	Allocates space
`HDfreespace`	Frees space
`HDstrncpy`	Copies a string from one location to another up to a given number of characters

Error Reporting

The HDF library incorporates the notion of an *error stack.* This allows much of the context to be known when trying to decipher an error message.

Error reporting is handled by the following routines:

`HEprint`	Prints out all of the errors on the error stack to a specified file
`HEclear`	Clears the error stack
`HERROR`	Reports an error Pushes the following information onto the error stack:

Error type
source file name
Line number and the name of the function reporting the error

`HEreport` Adds a text string to the description of the most recently reported error (only one text string per error)

Standard C does not enable the code inside a function to know the name of the function. Therefore, to use the macro `HERROR` to report errors, there must exist a variable `FUNC` which points to a string containing the name of the reporting function.

Other

The `Hsync` routine has been defined and implemented to synchronize a file with its image in memory. Currently it is not very useful because the HDF software includes no buffering mechanism and the two images are always identical. `Hsync` will become useful when buffering is implemented:

`Hsync` Synchronizes the stored version of an HDF file with the image in memory

Function Specifications

The terms IN: and OUT: are used as follows in this discussion:

IN:	Value as input parameter
OUT:	Value as output parameter

Opening and Closing Files

Hopen

```
int32 Hopen(char *path, int access, int16 ndds)
```

path	IN:	Name of file to be opened
access	IN:	DFACC_READ, DFACC_RDWR, DFACC_CREATE, DFACC_ALL, or DFACC_WRITE
ndds	IN:	Number of DDs in a block if this file needs to be created

Purpose: Provides an access path to an HDF file and reads all of the DD blocks in the file into primary memory.

Return value: Returns file ID if successful and FAIL (-1) otherwise.

Description: Opens an HDF file.

The following events occur on successful exit:

- File_rec members are filled in. (File_rec is an internal HDF structure containing information about the opened file.)
- The requested file is opened with the relevant permission.
- Information about DDs is set up in memory.
- The file headers and initial information are set up for new files.

Access privilege codes

HDF provides several constants for use as access privilege codes as listed below. Note that these constants are not bit-flags and should not be ORed together to combine access modes. Doing so may cause odd behavior and, in some cases, loss of data:

Recommended:

DFACC_READ	Open for read only. If file does not exist, error.
DFACC_RDWR	Open for read/write. If file does not exist, create it.
DFACC_CREATE	Force creation. If file exists, delete it, then open a new file for read/write (in the spirit of the UNIX System command clobber).

Others:

DFACC_ALL	Same as DFACC_RDWR (obsolete but still supported).
DFACC_WRITE	Same as DFACC_RDWR (obsolete but still supported).

`Hclose`

`intn Hclose(int32 *id*)`

id	IN:	The file ID of the file to be closed

Purpose: Closes the access path to the file.

Return value: Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description: *`id`* is first validated. If valid, the function closes the access path to the file.

If there are still access elements attached to the file, the error `DFE_OPENAID` is pushed onto the error stack and the file is not closed. This is a fairly common error when developing new interfaces. See the discussion of `Hendaccess` below for debugging hints.

Locating Elements for Access and Getting Information

`Hstartread`

`int32 Hstartread(int32 file_id, uint16 tag, uint16 ref)`

file_id	IN:	ID of file to attach access element to
tag	IN:	Tag to search for
ref	IN:	Reference number to search for

Purpose — Locates an existing data element with matching tag/ref and returns an access ID for reading it.

Return value — Returns access element ID if successful and FAIL (`-1`) otherwise.

Description — Searches the DDs for a particular tag/ref combination. If the search is successful, an access element is created, attached to the file, and positioned at the start of that data element; otherwise an error is returned. Searching on wildcards begins from the beginning of the DD list. Wildcards can be used for the tag or reference number (`DFTAG_WILDCARD` and `DFREF_WILDCARD`) and they match any values.

`Hnextread`

`intn Hnextread(int32 access_id, uint16 tag, uint16 ref, int origin)`

access_id	IN:	ID of a READ access element
tag	IN:	Tag to search for
ref	IN:	Reference number to search for
origin	IN:	Position at which to start searching

Purpose — Locates and positions a read access ID on next occurrence of tag/ref.

Return value — Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description — Searches for the next DD that fits the tag/ref. Wildcards apply. If *origin* is `DF_START`, searches from start of DD list; if *origin* is `DF_CURRENT`, searches from current position. Searching from the end of the file via `DF_END` is not yet implemented.

If the search is successful, then the access element is positioned at the start of that tag/ref; otherwise, the access ID is not modified.

Hstartwrite

`int32 Hstartwrite(int32 file_id, uint16 tag, uint16 ref, int32 length)`

file_id	IN:	ID of file to write to
tag	IN:	Tag to write to
ref	IN:	Reference number to write to
length	IN:	Length of the data element

Purpose — Creates or replaces data element with matching tag/ref.

Return value — Returns access element ID if successful and FAIL (`-1`) otherwise.

Description — Sets up an access element to write a data element. The DD list of the file is searched first; if the tag/ref is found, the data element can be modified. If an object with the corresponding tag/ref is not found, a new one is created.

Hendaccess

`int32 Hendaccess(int access_id)`

access_id	IN:	ID of access element to dispose of

Purpose — Disposes of access element for tag/ref.

Return value — Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description — Disposes of an access element. Only a finite number of access elements can be active at a given time, so it is important to call `Hendaccess` whenever you are done using an element.

When developing new interfaces, a common mistake is to fail to call `Hendaccess` for all of the elements accessed. When this happens, `Hclose` will return FAIL and the dump of the error stack (see `HEprint` below) will tell how many access elements are still active.

This can be a difficult problem to debug, as the low levels of the HDF library have no idea who or what opened an access element and forgot to release it. A tedious but effective means of debugging this problem is to annotate with comments the locations where the attached count of a file record is changed. This occurs in the files `hfile.c`, `hblocks.c`, and `hextelt.c`.

Hinquire

```
intn Hinquire(int32 access_id, int32 *pfile_id, uint16 *ptag,
      uint16 *pref, int32 *plength, int32 *poffset, int32 *pposn,
      int *paccess, int16 *pspecial)
```

access_id	IN:	Access element ID
pfile_id	OUT:	File ID
ptag	OUT:	Tag of the element pointed to
pref	OUT:	Reference number of the element pointed to
plength	OUT:	Length of the element pointed to
poffset	OUT:	Offset of element in the file
pposn	OUT:	Position pointed to within the data element
paccess	OUT:	Access type of this access element
pspecial	OUT:	Special code

Purpose: Returns access information for a data element.

Return value: Returns SUCCEED (0) if the access element points to some data element and FAIL (-1) otherwise.

Description: Inquires for the statistics of the data element pointed to by the access element. If a piece of information is not needed, a NULL can be sent in for that value. Convenience macros for calls to `Hinquire` (`HQuerypositon`, `HQuerylength`, etc.) are defined in `hdf.h`.

Hishdf

```
int32 Hishdf(char *path)
```

path	IN:	Name of file

Purpose: Determines whether a file is an HDF file.

Return value: Returns TRUE (non-zero) if file is an HDF file and FALSE (`0`) otherwise.

Description: The decision as to whether a file is an HDF file is based solely on the magic number stored in the first four bytes of an HDF file. `Hishdf` may sometimes identify a file as an HDF file that `Hopen` is unable to open (e.g., an HDF file with a corrupted DD list).

Note: `Hishdf` only determines whether a file is an HDF file. It does not verify that the file is readable.

Hnumber

```
int Hnumber(int32 file_id, uint16 tag)
```

file_id	IN:	File ID
tag	IN:	Tag to be counted

Purpose	Counts the number of occurrences of a tag in a file.
Return value	The number of occurrences of a tag in a file.

Hgetlibversion

```
Hgetlibversion(uint32 *majorv, uint32 *minorv, uint32 *release,
      char string[])
```

majorv	OUT:	Major version number
minorv	OUT:	Minor version number
release	OUT:	Release number
string	OUT:	Informational text string

Purpose	Gets version information for current HDF library.
Return value	Returns SUCCEED (`0`).
Description	Returns the version of the HDF library. The version information is compiled into the HDF library, so it is not necessary to have any open files for this function to execute.

Hgetfileversion

```
Hgetfileversion(uint32 file_id, uint32 *majorv, uint32 *minorv,
      uint32 *release, char *string)
```

file_id	IN:	File ID
majorv	OUT:	Major version number
minorv	OUT:	Minor version number
release	OUT:	Release number
string	OUT:	Informational text string

Purpose	Gets version information for an HDF file.
Return value	Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.
Description	Returns the HDF version information stored in the given file.

Reading and Writing Entire Data Elements

Hputelement

```
int Hputelement(int32 file_id, uint16 tag, uint16 ref, uint8 *data,
int32 length)
```

file_id	IN:	File ID
tag	IN:	Tag of data element to put
ref	IN:	Reference number of data element to put
data	IN:	Pointer to buffer
length	IN:	Length of data

Purpose: Adds or replaces an element in a file.

Return value: Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description: Writes a new data element or replaces an existing data element in a HDF file. Uses `Hwrite` and its associated routines.

Hgetelement

```
int Hgetelement(int32 file_id, uint16 tag, uint16 ref, uint8 *data)
```

file_id	IN:	ID of the file to read from
tag	IN:	Tag of data element to read
ref	IN:	Reference number of data element to read
data	OUT:	Buffer to read into

Purpose: Obtains the data referred to by the passed tag/ref.

Return value: Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description: Reads a data element from an HDF file and puts it into the buffer pointed to by `data`. The space allocated for the buffer is assumed to be large enough.

Note: `Hgetelement` assumes that the buffer is large enough to hold the data being read. It is the user's responsibility to prevent data loss by ensuring that this is the case.

Reading and Writing Part of a Data Element

`Hread`

`int32 Hread(int32 access_id, int32 length, uint8 *data)`

access_id	IN:	Read access element ID
length	IN:	Length of segment to read in
data	OUT:	Pointer to data array to read to

Purpose — Reads a portion of a data element.

Return value — Returns length of segment actually read if successful and FAIL (-1) otherwise.

Description — Reads in the next segment in the data element pointed to by the access element. `Hread` starts at the last position left by an `Hread` or `Hseek` call and reads any data that remains in the element up to *length* bytes. If the data element is too short (less than *length* bytes long), `Hread` reads to the end of the data element.

`Hwrite`

`int32 Hwrite(int32 access_id, int32 length, uint8 *data)`

access_id	IN:	Write access element ID
length	IN:	Length of segment to write
data	IN:	Pointer to data to write

Purpose — Writes next data segment to data element.

Return value — Returns length of segment successfully written and FAIL (`-1`) otherwise.

Description — Writes the data to the data element where the last `Hwrite` or `Hseek` stopped.

`Hwrite` starts at the last position left by an `Hwrite` or `Hseek` call, writes up to a specified number of bytes, and leaves the write pointer at the end of the data written. If the space reserved is less than the length to write, then only as much as can fit is written.

It is the user's responsibility to ensure that no two access elements are writing to the same data element. Note that a user can interlace writes to multiple data elements in the same file.

Hseek

`intn Hseek(int32 access_id, int32 offset, int origin)`

access_id	IN:	Access element ID
offset	IN:	Offset to seek to
origin	IN:	Position to seek from:
		`DF_START (0)` *offset* from beginning of data element
		`DF_CURRENT (1)` *offset* from current position
		`DF_END (2)` *offset* from end of data element

Purpose — Sets the access pointer to an offset within a data element. The next time `Hread` or `Hwrite` is called, the read or write occurs from the new position.

Return value — Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.

Description — Sets the position of an access element in a data element so that the next `Hread` or `Hwrite` will start from that position. *origin* determines the position from which *offset* should be counted.

This routine fails if the access element is not associated with a data element or if the position sought is outside of the data element.

Seeking from the end of a data element is not currently supported.

Manipulating Data Descriptors

Hdupdd

```
int Hdupdd(int32 file_id, uint16 tag, uint16 ref, uint16 old_tag,
      uint16 old_ref)
```

file_id	IN:	File ID
tag	IN:	Tag of new data descriptor
ref	IN:	Reference number of new data descriptor
old_tag	IN:	Tag of data descriptor to duplicate
old_ref	IN:	Reference number of data descriptor to duplicate

Purpose	Generates new references to data that is already referenced from somewhere else.
Return value	Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.
Description	Duplicates a data descriptor so that the new tag/ref points to the same data element pointed to by the old tag/ref.

Hdeldd

```
int Hdeldd(int32 file_id, uint16 tag, uint16 ref)
```

file_id	IN:	File ID
tag	IN:	Tag of data descriptor to delete
ref	IN:	Reference number of data descriptor to delete

Purpose	Deletes a tag/ref from the list of DDs.
Return value	Returns SUCCEED (`0`) if successful and FAIL (`-1`) otherwise.
Description	Deletes the data descriptor of tag/ref from the DD list of the file. This routine is unsafe and may leave a file in a condition that is not usable by some routines. Use with care.

Hnewref

`uint16 Hnewref(int32 *file_id*)`

file_id	IN:	File ID

Purpose: Returns the next available reference number.

Return value: Returns the reference number if successful and `0` otherwise.

Description: Returns a reference number that can be used with any tag to produce a unique tag/ref. Successive calls to `Hnewref` will generate a strictly increasing sequence until the highest possible reference number has been returned; then `Hnewref` will return unused reference numbers starting from `1`.

Creating Special Data Elements

HLcreate

```
int32 HLcreate(int32 file_id, uint16 tag, uint16 ref,
      int32 block_length, int32 number_blocks)
```

file_id	IN:	File ID
tag	IN:	Tag of new data element (or object)
ref	IN:	Reference number of new data element (or object)
block_length	IN:	Length of blocks to be used
number_blocks	IN:	Number of blocks to use per linked block record

Purpose: Creates a new linked block special data element.

Return value: Returns access ID for special data element if successful and FAIL (-1) otherwise.

Description: Appending to existing HDF elements was a problem prior to HDF Version 3.2 because HDF objects had to be stored contiguously. When appending, the HDF library forced the user to delete the existing element and rewrite it at the end of the file. HDF Version 3.2 introduced the concept of linked blocks, which allow unlimited appending to existing elements without copying over existing data.

This routine can be used to create an object with the given tag/ref as a linked block element or to promote an existing element to be stored in linked blocks.

Initially, a table is set up to accommodate *number_blocks* linked blocks for the specified data object. Each block has *block_length* bytes. If an existing object is being promoted, *block_length* does not have to be the same size as the original element.

`HLcreate` returns an active access ID with write permission to the linked block element.

HXcreate

```
int32 HXcreate(int32 file_id, uint16 tag, uint16 ref,
      char *extern_file_name)
```

file_id	IN:	file record ID
tag	IN:	Tag of the special data element to create or promote
ref	IN:	Reference number of the special data element to create/promote
extern_file_name	IN:	name of the external file to use for the data element

Purpose: Creates a new external file special data element.

Return value: Returns access ID for special data element if successful and FAIL (-1) otherwise.

Description: Creates a new element in an external file or promotes an existing element to be stored in an external file. If an existing element is to be promoted, it is deleted (using Hdeldd) from the original file and copied into the new external file.

Distributing a single object over multiple external files is not currently supported. In addition, one cannot place multiple objects in the same external file.

This routine returns an active access ID with write permission to the external element.

Development Routines

HDgettagname

`char *HDgettagname(uint16 tag)`

tag	IN:	Tag to look up

Purpose Gets a meaningful description of a tag.

Return value Returns a pointer to a string describing this tag or NULL if the tag is unknown.

Description To reduce the amount of duplicated code, this routine can be used to map a tag to a character string containing the name of the tag.

The string returned by this routine is guaranteed to be 30 characters or less.

HDgetspace

`void *HDgetspace(uint32 qty)`

qty	IN:	Number of bytes to allocate

Purpose Allocates space.

Return value If successful, returns a pointer to space that was allocated; otherwise returns NULL .

Description Uses an appropriate allocation routine on the local machine to get space.

HDfreespace

`void *HDfreespace(void *ptr)`

ptr	IN:	Pointer to previously-allocated space that is to be freed

Purpose Frees space.

Return value Returns NULL.

Description Uses an appropriate routine on the local machine to free space. This routine is platform dependent.

HDstrncpy

```
char *HDstrncpy(register char *dest, register char *source,
      int32 length)
```

dest	OUT:	Pointer to area to copy string to
source	IN:	Pointer to area to copy string from
length	IN:	Maximum number of bytes to copy

Purpose Copies a string with maximum length `length`.

Return value Returns address of `dest`.

Description Creates a string in `dest` that is at most `length` characters long. The number of characters must *include* the NULL terminator for historical reasons. Hence, if you are working with the string `Foo,` you must call this copy function with the value `4` (three characters plus the NULL terminator) in `length`.

Error Reporting

HEprint

```
void HEprint(FILE *stream, int32 level)
```

stream	IN:	Stream to print error messages on
level	IN:	Level of the error stack to print

Purpose: Prints information on the error stack.

Return value: Has no return value.

Description: Prints information on reported errors. If `level` is zero, all of the errors currently on the error stack are printed. Output from this function is sent to the file pointed to by `stream`.

The following information printed:
- An ASCII description of the error
- The reporting routine
- The reporting routine's source file name
- The line at which the error was reported

If the programmer has supplied extra information by means of `HEreport`, this information is printed as well.

HEclear

```
void HEclear(void)
```

Purpose: Clears all information on reported errors off of the error stack.

Return value: Has no return value.

Description: Clears all of the information off of the error stack.

HERROR

```
void HERROR(int16 number)
```

number	IN:	Error number

Purpose: Reports an error.

Return value: Has no return value.

Description: Reports an error. Any function calling `HERROR` must have a variable `FUNC` which points to a string containing the name of the function.

`HERROR` is implemented as a macro.

HEreport

void HEreport(char **format*, ...)

format	IN:	`printf`-style format and arguments

Purpose: Provides extra information to the error reporting routines.

Return value: Has no return value.

Description: Provides further annotation to an error report. Only one such annotation is remembered for each error report. The arguments to this routine follow the style of `printf`.

Consider the following example from `hfile.c`:

```
char *FUNC = "Hclose";
....
if (file_rec->attach > 0) {
    file_rec->refcount++;
    HERROR(DFE_OPENAID);
    HEreport("There are still %d active aids attached", file_rec->attach);
    return FAIL;
}
```

Other

Hsync

```
int Hsync(int32 file_id)
```

file_id	IN:	ID of the file to synchronize

Purpose Synchronizes on-disk HDF file with image in memory.

Return value Returns SUCCEED.

Description `Hsync` is not included in the current HDF library release because the on-disk representation of an HDF file is always the same as its in-memory representation. `Hsync` will be provided when future releases implement buffering schemes.

Chapter 4 Sets and Groups

Chapter Overview

This chapter discusses the roles of the following sets and groups in organizing data stored in an HDF file:

- Raster image sets (RIS)
  - Raster image groups (RIG)
- Scientific data sets (SDS)
  - Scientific data groups (SDG)
  - Numeric data groups (NDG)
  - SDG-like NDGs
- Vsets
  - Vgroups
- Raster-8 sets (obsolete)

This chapter introduces several tags used in support of sets and groups. All of these tags are fully described in Chapter 6, “Tag Specifications,” and are listed in the table in Appendix A, “NCSA HDF Tags.”

Data Sets

HDF files frequently contain several closely related data objects. Taken together, these objects form a *data set* which serves a particular user requirement. For example, five or six data objects might be used to describe a raster image; eight or more data objects might be used to describe the results of a scientific experiment.

The HDF mechanism for specifying and controlling data sets is the *group*. The data element of a group consists of a single record listing the tag/refs for all the objects contained in the data set. For example, the raster image groups described in the following sections each contain three tag/refs that point to three data objects that, taken as a set, fully describe an 8-bit raster image.

Types of Sets

The current HDF implementation supports three kinds of sets:

Raster image set
: A set containing a raster image and descriptive information such as the image dimensions and an optional color lookup table

Scientific data set
: A set containing a multidimensional array and information describing the data in the array

Vset
A general grouping structure containing any kinds of HDF objects that a user wishes to include

Each HDF set is defined with a minimum collection of data objects that will make sense when the set is used. For example, every raster image set must contain at least the following data objects:

Raster image group
The list of the members of the set

Image dimension record
The width, height, and pixel size of the raster image

Raster image data
The pixel values that make up the image

In addition to the required objects, a set may include optional data objects. An 8-bit raster image set, for instance, often contains a palette, or color lookup table, which defines the red, green, and blue values associated with each pixel in the raster image.

Calling Interfaces for Sets

NCSA provides calling interfaces for all the HDF sets that it supports. These interfaces provide routines for reading and writing the data associated with each set. The libraries currently supported by NCSA are callable from either C or FORTRAN programs.

In addition to the libraries, a growing number of command-line utilities are available to manipulate sets. For example, a utility called `r8tohdf` converts one or more raw raster images to HDF 8-bit raster image set format.

The calling interfaces are described in the document *NCSA HDF Calling Interfaces and Utilities* for Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* for Version 3.3.

Groups

As discussed above, HDF data objects are frequently associated as sets. But without some explicit identifying mechanism, there is often no way to tie them together. To address this problem, HDF provides a grouping mechanism called a *group*. A group is a data object that explicitly identifies all of the data objects in a set.

Since a group is just another type of data object, its structure is like that of any other data object; it includes a DD and a data element. But instead of containing the pixel values for a raster image or the dimensions of an array, a group data element contains a list of tag/refs for the data objects that make up the corresponding set.

A *group tag* can be defined for any set. For instance, the *raster image group* tag (RIG, `DFTAG_RIG`) is used to identify members of raster image sets; the RIG data element lists the tag/refs for a particular raster image set.

An Example

Suppose that the two images shown in Figure 1.5, "Physical Representation of Data Objects," are organized into two sets with group tags. Since they are raster images, they may be stored as RIGs. Figure 4.1 illustrates the use of RIGs with these images.

Figure 4.1 Physical Organization of Sample RIG Groupings

Offset	Item	Contents
0	FH	`0e031301` *(HDF magic number)*
4	DDH	`10 0L`
10	DD	`DFTAG_FID 1 130 4`
22	DD	`DFTAG_FD 1 134 41`
34	DD	`DFTAG_LUT 1 175 768`
46	DD	`DFTAG_ID 1 943 4`
58	DD	`DFTAG_RI 1 947 240000`
70	DD	`DFTAG_ID 2 240947 4`
82	DD	`DFTAG_RI 2 240951 240000`
94	DD	`DFTAG_RIG 1 480951 12`
106	DD	`DFTAG_RIG 2 480963 12`
118	DD	`DFTAG_NULL` *(Empty)*
130	Data	`sw3`
134	Data	`solar wind simulation: third try. 8/8/88`
175	Data	`...` *(Data for image palette)*
943	Data	`400, 600 ...` *(Data for 1st image dimension record)*
947	Data	`...` *(Data for 1st raster image)*
240947	Data	`400, 600 ...` *(Data for 2nd image dimension record)*
240951	Data	`...` *(Data for 2nd raster image)*
480951	Data	`DFTAG_IP8/1, DFTAG_ID/1, DFTAG_RI/1` *(Tag/refs for 1st RIG)*
480963	Data	`DFTAG_IP8/1, DFTAG_ID/2, DFTAG_RI/2` *(Tag/refs for 2nd RIG)*

The file depicted in Figure 4.1 contains the same raster image information as the file in Figure 1.5, but the information is organized into two sets. Note that there is only one palette (`DFTAG_IP8/1`) and that it is included in both groups.

General Features of Groups

Figure 4.1 also illustrates a number of important general features of groups:

- The contents of a group must be consistent with one another. Since the palette (`DFTAG_IP8`) is designed for use with 8-bit images, the image must be an 8-bit image.
- An application program can easily process all of the images in the file by accessing the groups in the file. The non-RIG information in the example can be used or ignored, depending on the needs and capabilities of the application program.
- There is usually more than one way to group sets. For example, an extra copy of the image palette (`DFTAG_IP8`) could have been stored

in the file so that each grouping would have its own image palette. That is not necessary in this instance because the same palette is to be used with both images. On the other hand, there are two image dimension records in this example, even though one would suffice.

- Group status does not alter the fundamental role of an HDF object; it is still accessible as an individual data object despite the fact that it also belongs to a larger set.

- A group provides an index of the members of a set. There is nothing to prevent the imposition of other groupings (indexes) that provide a different view of the same collection of data objects. In fact, HDF is designed to encourage the addition of alternate views.

The following sections formally describe raster image sets (RIS), scientific data sets (SDS), Vsets, and several related groups. The last section of this chapter discusses an obsolete structure known as the raster-8 set.

Raster Image Sets (RIS)

The raster image set (RIS) provides a framework for storing images and any number of optional image descriptors. An RIS always contains a description of the image data layout and the image data. It may also contain color look-up tables, aspect ratio information, color correction information, associated matte or other overlay information, and any other data related to the display of the image.

Raster Image Groups (RIG)

Tying everything together is the raster image group (RIG, see Figure 4.1 and the related discussion for an example). An RIG contains a list of tag/refs that point in turn to the data objects that make up and describe the image.

The number of entries in an RIG is variable and most of the descriptive information is optional. Complex applications may include references to image-modifying data, such as the color table and aspect ratio, along with the reference to the image data itself. Simple applications may use simple application-level calls and ignore specialized video production or film color correction parameters.

NCSA currently supports two RIG calling interfaces: *RIS8* and *RIS24*. These interfaces are described in the document *NCSA HDF Calling Interfaces and Utilities* for Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* for Version 3.3.

RIS Tags

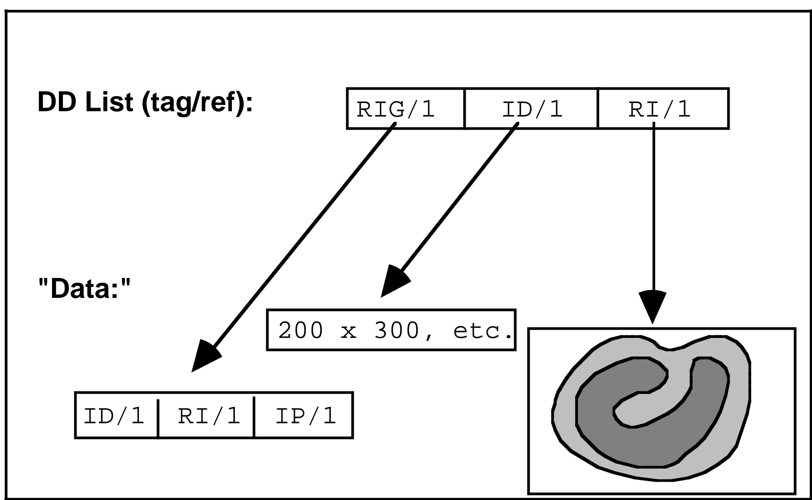
RIS implementations must fully support all of the tags presented in Table 4.1.

Table 4.1 RIS Tags

Tag	Contents of Data Element
`DFTAG_RIG`	Raster image group
`DFTAG_ID`	Image dimension record
`DFTAG_RI`	Raster image data

With these tags, images can be stored and read from HDF files at any bit depth, with several different component ordering schemes. As illustrated in Figure 4.1, the RIG tag points to the collection of tag/refs that fully describe the RIS. The data element attached to the tag `DFTAG_ID` specifies the dimensions of the image, the number type of the elements that make up its pixels, the number of elements per pixel, the interlace scheme used, and the compression scheme used, if any. The data element attached to the tag `DFTAG_RI` contains the actual raster image data.

Figure 4.1 RIS Tags



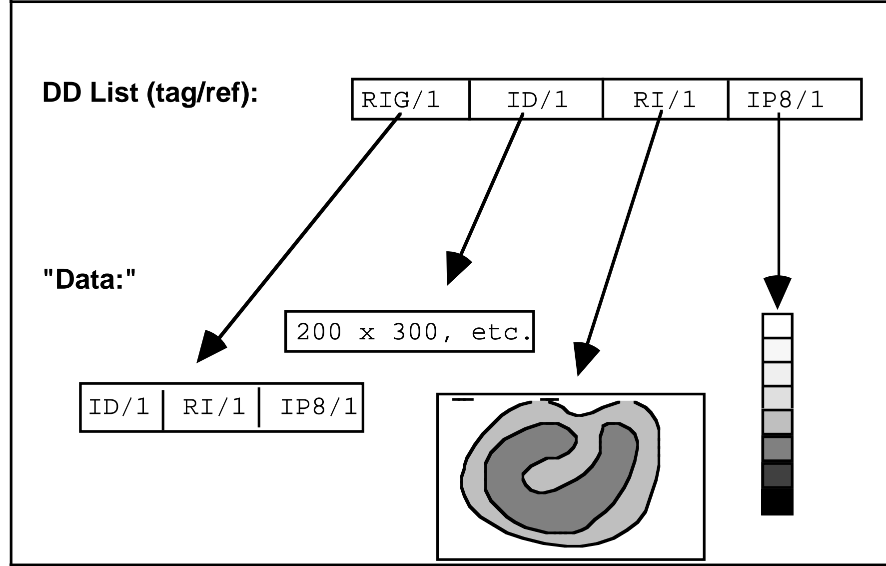
The tags listed in Table 4.2 identify optional RIS information such as color properties and aspect ratio. Note that the RI interface supports only `DFTAG_LUT` at this time; the other tags in Table 4.2 are defined but the interfaces have not been implemented.

Table 4.2 Optional RIS Tags

Tag	Contents of Data Element
DFTAG_XYP	XY position of image
DFTAG_LD	Look-up table dimension record
DFTAG_LUT	Color look-up table for non true-color images
DFTAG_MD	Matte channel dimension record
DFTAG_MA	Matte channel data
DFTAG_CCN	Color correction factors
DFTAG_CFM	Color format designation
DFTAG_AR	Aspect ratio
DFTAG_MTO	Machine-type override

Figure 4.2 illustrates the structure of an RIS that contains an image palette (DFTAG_IP8).

Figure 4.2 RIS Tags for Sets Containing a Palette



Raster Image Compression

HDF currently supports two raster image compression tags:

DFTAG_RLE	Run-length encoding
DFTAG_IMCOMP	Aerial averaging
DFTAG_JPEG	JPEG compression

RIG support does not require support for all compression tags. Be sure to provide a suitable error message to the user when an unknown compression tag is encountered.

Since new forms of data compression can be added to HDF raster images, incompatibilities can arise between old libraries and files created by newer libraries. For example, HDF Version 3.3 includes JPEG compression for images. A JPEG-compressed raster image in a file created by an HDF Version 3.3 library cannot be read by an HDF Version 3.2 library.

Scientific Data Sets

The scientific data set (SDS) provides a framework for storing multidimensional arrays of data with descriptive information that enhances the data. Current specifications support the following types of numbers in SDS arrays.

- 8-bit, 16-bit, and 32-bit signed and unsigned integers
- 32-bit and 64-bit floating point numbers

Data in an SDS can be stored either as two's complement big endian integers, as IEEE Standard floating point numbers, or in *native mode*, the format used by the machine from which they were written.

The user interface for storing and retrieving SDSs is fully described in the document *NCSA HDF Calling Interfaces and Utilities* for Versions 3.2 and earlier and in the *NCSA HDF User’s Guide* and *NCSA HDF Reference Manual* for Version 3.3.

Backward and forward compatibility

One of NCSA’s concerns in HDF development is always to maximize backward and forward compatibility; as much as possible, any application written to use HDF should be able to read data files written with an older or a newer version of the libraries. To maximize this compatibility, NCSA had to consider the following factors in upgrading the SDS capabilities:

- Support for future variations (e.g., new number types, data compression, and new physical arrangements for SDS storage)

- Older versions of the library should be able to read new data files if the data itself can be interpreted by the older version. To do so, the older version must be able to determine whether the data in a given data object will be comprehensible to it. For example, if a newly created file contains 32-bit IEEE floating point or Cray floating point data objects, older versions of the library should be able determine that fact then read and interpret the data.

- New libraries must be able to read and interpret files created by older versions.

Unfortunately, such compatibility concerns yield an SDS structure somewhat more complex than would otherwise be the case. Two examples illustrate the problem:

- HDF 3.2 development had to accommodate the fact that HDF Version 3.1 and previous versions only supported 32-bit IEEE floating-point numbers and Cray floating point numbers in SDSs. SDSs in HDF versions since Version 3.2 support 8-bit, 16-bit, and 32-bit signed and unsigned integers, 32-bit and 64-bit floating-point numbers, and the local machine format (*native mode*) for all supported architectures.

- HDF 3.3 includes support for the netCDF data model, which involved the creation of an entire new structure for supporting netCDF objects, based on Vgroups and Vdatas. At the same time, a goal of HDF 3.3 was to harmonize the SDS and the netCDF data

model, which was best accomplished by storing SDS objects in the same way that netCDF objects are stored. In order to maintain backward compatibility, two structures had to be created for every SDS or netCDF object: one that could be recognized by older HDF libraries, and the new structure.

In the following sections we describe how the first problem was solved. A later issue of this manual will describe how the second problem was addressed.

Internal Structures

The SDS capability was substantially enhanced for HDF Version 3.2. Previous versions employed a structure known as a *scientific data group* (SDG); Version 3.2 and subsequent versions use the *numeric data group* (NDG). To accommodate the enhanced structure and to remain compatible with previous releases, the current HDF library supports the following scientific and numerical data groups:

SDGs
: Created by old libraries and containing 32-bit IEEE and Cray floating-point data.

NDGs
: Created by the newer libraries (Version 3.2 and later) and containing any acceptable floating-point or non-floating-point data. This data group will not be recognized by old libraries.

SDG-like NDGs
: Created by the new library and containing IEEE 32-bit floating-point data only. The old libraries will recognize and interpret these numerical data groups correctly.

The NDG structure supports 8-bit, 16-bit, and 32-bit signed and unsigned integers, and 32-bit and 64-bit floating-point numbers. It also supports *native mode*, data sets written to HDF files in the local machine format.

The following sections describe the SDG, NDG, and SDG-like NDG structures.

SDG Structures

SDGs must contain at least the data objects listed in Table 4.3.

Table 4.3 Required SDG Tags

Tag	Contents of Data Element
DFTAG_SDG	Scientific data group.
DFTAG_SDD	Dimension record for array-stored data. Includes the rank (number of dimensions), the size of each dimension, and the tag/refs representing the number type of the array data and of each dimension. All SDG number types are 32-bit IEEE floating-point.
DFTAG_SD	Scientific data.

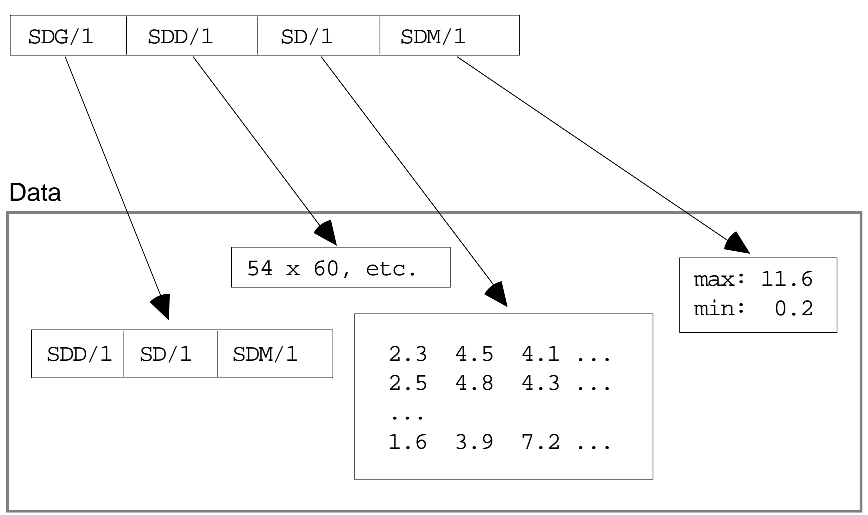
In addition to the required data objects listed above, SDGs may contain any of the objects listed in Table 4.4. Note that the optional data objects are the same for SDGs, NDGs, and SDG-like NDGs; the only differences are the number types that may be used.

Table 4.4 Optional SDG, NDG, and SDG-like NDG Tags

Tag	Contents of Data Element
DFTAG_SDS	Scales of the different dimensions. To be used when interpreting or displaying the data (32-bit floating point numbers only for SDGs and SDG-like NDGs).
DFTAG_SDL	Labels for all dimensions and for the data. Each of the dimension labels can be interpreted as an independent variable; the data label is the dependent variable.
DFTAG_SDU	Units for all dimensions and for the data.
DFTAG_SDF	Format specifications to be used when displaying values of the data.
DFTAG_SDM	Maximum and minimum values of the data. (32-bit floating point numbers only for SDGs and SDG-like NDGs.)
DFTAG_SDC	Coordinate system to be used when interpreting or displaying the data.

As illustrated in Figure 4.3, the SDG tag points to the collection of tag/refs that define the SDG.

Figure 4.3 SDG Structure



NDG Structures

NDGs must contain at least the data objects listed in Table 4.5

Table 4.5 Required NDG Tags

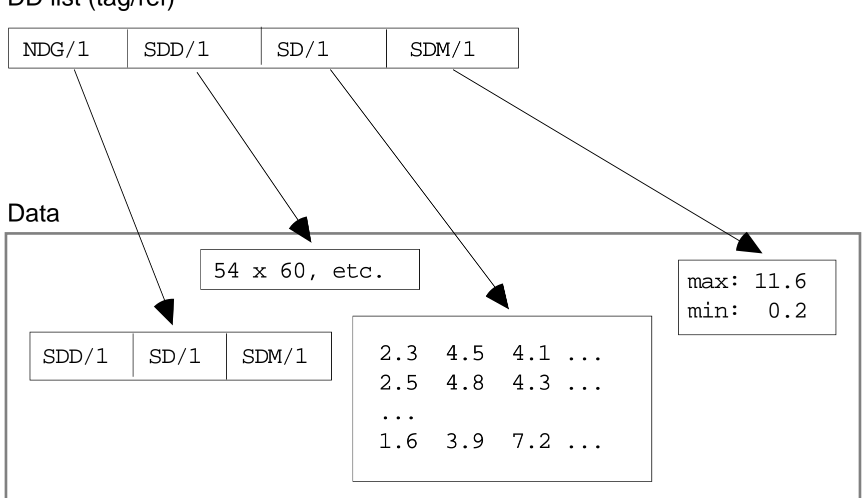
Tag	Contents of Data Element
DFTAG_NDG	Numerical data group.
DFTAG_SDD	Dimension record for array-stored data. Includes the rank (number of dimensions), the size of each dimension, and the tag/refs representing the number types of the data and of each dimension.

	In HDF 3.2 , the number types of dimension scales must be the same as that of the array-stored data. Later implementations allow dimension scales to be typed separately.
DFTAG_SD	Scientific data.
DFTAG_NT	Number type of the data set. Default is the most recent DFSDsetNT() setting. If DFSDsetNT() has not been called, the default will be 32-bit IEEE floating-point.

In addition to these required data objects, an NDG may contain any of the data objects listed in Table 4.4, "Optional SDG, NDG, and SDG-like NDG Tags."

As illustrated in Figure 4.4, the basic NDG and SDG structures are identical. The first clue to the difference is that the NDG tag replaces the SDG tag. This is a flag to prevent older libraries from stumbling over the more important difference; the NDG data element can accommodate data that pre-Version 3.2 libraries cannot interpret. The new tag ensures that older libraries will not recognize the data object and thus will not try to interpret the new data types. For example, NDG data can include number types or a data compression scheme that a pre-Version 3.2 library will not recognize.

Figure 4.4 NDG Structure



SDG-like NDG Structures

As we have said earlier,

- SDGs, the SDS grouping structure available prior to HDF Version 3.2, could include only 32-bit floating point and Cray floating point numbers.
- NDGs, available since Version 3.2, can include 8-bit, 16-bit, and 32-bit signed and unsigned integers, and 32-bit and 64-bit floating point numbers.
- SDG-like NDGs, also available since Version 3.2, distinguish SDSs that can still be read by the older versions of the library.

This backward compatibility is achieved by examining every SDS that is written to an HDF file. If the SDS is compatible with older libraries,

it is written to the file with both SDG and NDG structures. If it is not compatible with older libraries, only the NDG structure is used.

Table 4.6 lists the objects that SDG-like NDGs must contain.

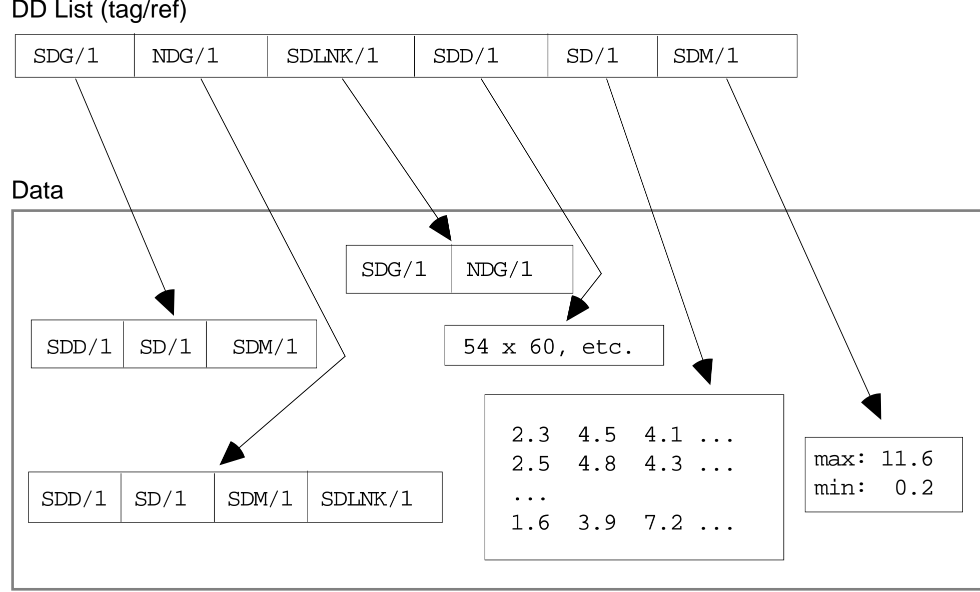
Table 4.6 Required SDG-like NDG Tags

Tag	Contents of Data Element
DFTAG_NDG	Numerical data group.
DFTAG_SDG	Scientific data group.
DFTAG_SDLNK	The NDG and SDG linked to the scientific data set in this group.
DFTAG_SDD	Dimension record for array-stored data. Includes the rank (number of dimensions), the size of each dimension, and the tag/refs representing the number types of the data and of each dimension. In an SDG-like NDG, the number types are all 32-bit IEEE floating-point.
DFTAG_SD	Scientific data.

SDG-like NDGs can include the same optional data objects as described for SDGs and NDGs in Table 4.4, "Optional SDG, NDG, and SDG-like NDG Tags."

Figure 4.5 illustrates the SDG-like NDG structure.

Figure 4.5 SDG-like NDG Structure



Compatibility with Future NDG Structures

Future HDF releases will probably support additional optional SDS features. These features will fall into the following categories:

Optional and compatible features
: Optional features that are compatible with older HDF versions even though they may not be supported in the older libraries.

For example, a new time stamp attribute might be added. The time stamp would not be understood by older libraries, but it would not render them unable to read the SDS data either

Optional and incompatible features
: Optional new features that may render the data unreadable by older HDF libraries.

For example, a compression attribute could be added. Older HDF libraries that contain no compression routines would not be able to read the compressed data.

A tag numbering convention has been developed to address this problem:

Required tags
: These tags are listed in Table 4.3, “Required SDG Tags,” Table 4.5, “Required NDG Tags,” and Table 4.6, “Required SDG-like NDG Tags.” All SDSs must contain all of the tags in at least one of these sets. (See Chapter 6, “Tag Specifications,” for the assigned tag numbers.)

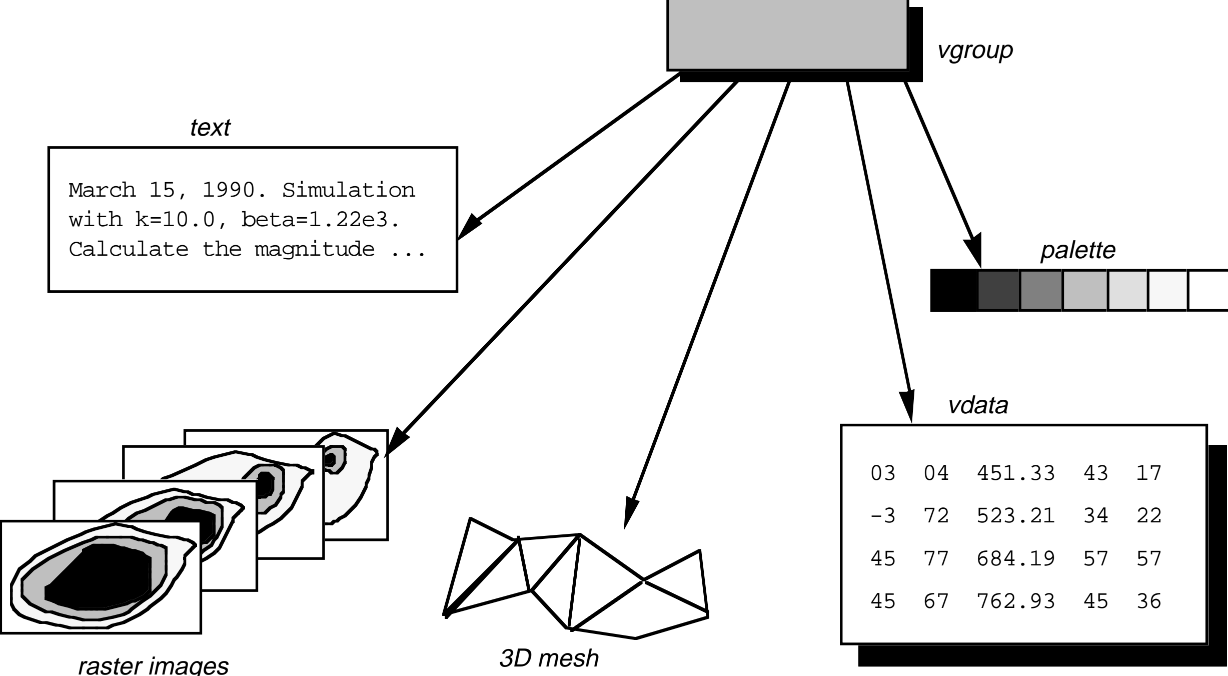
Optional-incompatible tags
: Tags for new SDS features that might render the data set unreadable by older libraries are each assigned a number `t` that falls in a special range determined by the constants `DFTAG_EREQ` and `DFTAG_BREQ`. That is, `t` must have a value such that `DFTAG_EREQ < t < DFTAG_BREQ`. When old software encounters a tag in this range that it is not able to interpret, it should not process the group.

Optional-compatible tags
: These tags can have any valid tag number not allocated to one of the other two categories.

Vsets, Vdatas, and Vgroups

Vsets, Vdatas, and Vgroups enable users to create their own grouping structures. Unlike RIGs, SDGs, and NDGs, HDF imposes no required structure; they are implemented almost entirely at the user level and are not specified in detail in HDF or in this document.* The only specifications define `DFTAG_VG`, `DFTAG_VH`, and `DFTAG_VS` and the formats of their respective data elements. A detailed discussion similar to that for the other grouping structures is, therefore, inappropriate here. Detailed information regarding the `DFTAG_VG`, `DFTAG_VH`, and `DFTAG_VS` tags can be found in Chapter 6, "Tag Specifications." Conceptual and usage information can be found in the document *NCSA HDF Vset Version 2.0* for HDF Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and the *NCSA HDF Reference Manual* for HDF Version 3.3.

Figure 4.6. Illustration of a Vset



An HDF Vset can contain any logical grouping of HDF data objects within an HDF file. Vsets resemble the UNIX file system in that they impose a basically hierarchical structure but also allow cross-linked data objects. Unlike SDSs and RISs, Vsets have no prespecified content or structure; users can use them to create structural relationships among HDF objects according to their needs. Figure 4.6 illustrates a Vset.

A Vset is identified by a *Vgroup*, an HDF object that contains information about the members of the Vset. The tag `DFTAG_VG` identifies the Vgroup which contains the tag/refs of its members, an

* Specialists in various fields are developing application program interfaces (APIs) that are becoming accepted standard interfaces within their fields. Since these APIs are implemented with high level HDF functionality and using the standard HDF user interface, they are user-level applications from the HDF development team's point of view. From the final enduser's point of view, however, these APIs create a new level of user interface. When necessary, technical specifications for these APIs and the associated interfaces will be presented by the specialized developers.

optional user-specified name, an optional user-specified class, and fields that enable the Vgroup to be extended to contain more information.

The only required Vgroup tag is the tag that defines the Vgroup itself.

Table 4.7 The Vgroup Tag

Tag	Contents of Data Element
DFTAG_VG	Vgroup

Vgroups are fully described in the document *NCSA HDF Vset, Version 2.0* for Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* for Version 3.3.

The Raster-8 Set (Obsolete)

Current HDF versions use the raster image set (RIS) to manage raster images. But before the RIS was implemented, a simpler, less flexible set called the *raster-8 set* was used for storing 8-bit raster images. This set is no longer supported in the HDF software, although it may turn up in some older HDF files.*

Raster-8 Sets

The *raster-8 set* is defined by a set of tags that provide the basic information necessary to store 8-bit raster images and display them accurately without requiring the user to supply dimensions or color information. The raster-8 set tags are listed in Table 4.9.

Table 4.9 Raster-8 Set Tags

Tag	Contents of Data Element
DFTAG_RI8	8-bit raster image data
DFTAG_CI8	8-bit raster image data compressed with run-length encoding
DFTAG_II8	IMCOMP compressed image data
DFTAG_ID8	Image dimension record
DFTAG_IP8	Image palette data

Software that does not support DFTAG_CI8 or DFTAG_II8 must provide appropriate error indicators to higher layers that might expect to find these tags.

Compatibility Between Raster-8 and Raster Image Sets

To maintain backward compatibility with raster-8 sets, the RIS interface stores tag/refs for both types of sets. For example, if an image is stored as part of a raster image set, there is one copy each of the image dimension data, the image data, and the palette data. But there were two sets of tag/refs pointing to each data element: one for the RIS and one for the raster-8 set. The image data, for example, is associated with the tags DFTAG_RI8 and DFTAG_RI.

Note: Raster-8 set support will not be maintained in future HDF releases.

Note that future HDF releases will phase out support for the raster-8 set. Therefore, new software should not expect to find both raster-8 and RIS structures supporting 8-bit raster images. Eventually, only RIS structures will be supported.

* In fact, during the first three years that RIS was used, the HDF software stored raster images in both RIS and raster-8 sets.

Chapter 5 Annotations

Chapter Overview

This chapter introduces annotations, HDF data objects used to annotate HDF files and objects.

The tags introduced in this chapter are fully described in Chapter 6, “Tag Specifications,” and are listed in the table in Appendix A, “Tags and Extended Tag Labels.”

General Description

It is often useful to attach a text annotation to an HDF file or its contents and to store that annotation in the same HDF file. HDF provides this capability through the *annotation* data object.

The data element of an annotation is a sequence of ASCII characters that can be associated with any of three types of objects:

- The file itself
- An individual HDF data object in the file
- A tag that identifies a data element

The current annotation interface supports only the first two.

Annotations come in two forms:

Label	A short, NULL-terminated string. Labels may include no embedded NULLs.
Description	A longer and more complex body of text of a pre-defined length. Descriptions may contain embedded NULLs.

Annotations are never required; they are used strictly at the discretion of the creator or user of an HDF file.

Table 5.1 shows the currently defined annotation types and their assigned tags.

Table 5.1 Annotation Tags

	Label Types	Description Types
File annotations	DFTAG_FID	DFTAG_FD
Object annotations	DFTAG_DIL	DFTAG_DIA
Tag annotations	DFTAG_TID	DFTAG_TD

The annotation interface is fully described in the document *NCSA HDF Calling Interfaces and Utilities* for Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* for Version 3.3

File Annotations

Any HDF file can include label annotations (DFTAG_FID) and/or description annotations (DFTAG_FD). The file annotation interface routines provided in the HDF software read and write file labels and file descriptions.

Object Annotations

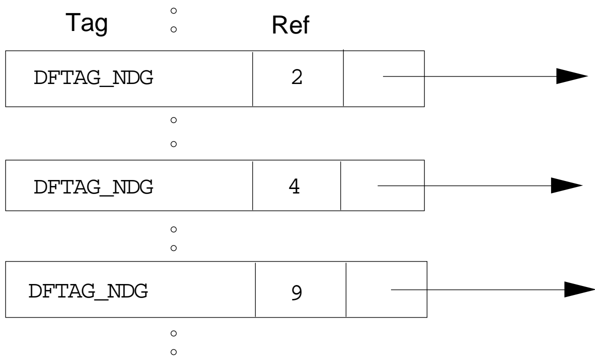
HDF data object annotation is complicated by the fact that you must uniquely identify the object being annotated. Since a tag/ref uniquely identifies a data object, the data object that a particular annotation refers to can be identified by storing the object's tag and reference number with the annotation.

Note that an HDF annotation is itself a data object, so it has its own DD. This DD has a tag/ref that points to the data element containing the annotation. The annotation data element contains the following information:

- The tag of the annotated object
- The reference number of the annotated object
- The annotation itself

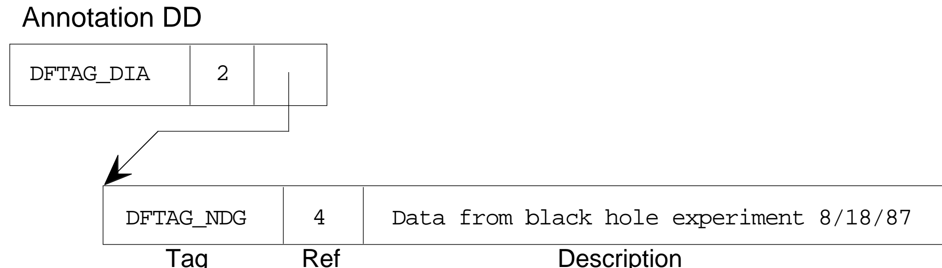
For example, suppose you have an HDF file that contains three scientific data sets (SDSs). Each SDS has its own DD consisting of the SDS tag DFTAG_SDG and a unique reference number, as illustrated in Figure 5.1.

Figure 5.1 Three SDS Tag/refs



Suppose you wish to attach the following annotation to the second SDS: "Data from black hole experiment 8/18/87." This text will be stored in a description annotation data object. The data element will include the tag/ref, DFTAG_NDG/4, and the annotation itself. Figure 5.2 illustrates the annotation data object.

Figure 5.2 Sample Annotation Data Object



Getting Reference Numbers for Object Annotations

To use annotation routines, you need to know the tags and reference numbers of the objects you wish to annotate.

The following routines return the most recent reference number used in either reading or writing the specified type of data object:

`DFSDlastref`	SDS data objects
`DFR8lastref`	RIS data objects
`DFPlastref`	Palettes
`DFANlastref`	Annotations

Reference numbers for other objects can be obtained with the routine `Hfindnextref`, a general purpose HDF routine that searches an HDF file sequentially for reference numbers associated with a given tag.

These routines are described in the document *NCSA HDF Calling Interfaces and Utilities* for Versions 3.2 and earlier and in the *NCSA HDF User's Guide* and *NCSA HDF Reference Manual* for Version 3.3.

Chapter 6 Tag Specifications

Chapter Overview

This chapter addresses issues related to HDF tags and the data they represent. The first section provides general information about tags and their interpretation. The remainder of the chapter contains a complete list of tags supported by NCSA HDF Version 3.3 and detailed tag specifications.

The HDF Tag Space

As discussed in Chapter 1, "The Basic Structure of HDF Files," 16 bits are allotted for an HDF tag number. This provides for 65535 possible tags, ranging from 1 to 65535; zero (0) is not used. This tag space is divided into three ranges:

1	–	32767	Reserved for NCSA-supported tags
32768	–	64999	Set aside as user-definable tags
65000	–	65535	Reserved for expansion of the format

No restrictions are placed on the user-definable tags. Note that tags from this range are not expected to be unique across user-developed HDF applications.

The rest of this chapter is devoted to the NCSA-supported tags in the range 1 to 32767.

Extended Tags and Alternate Physical Storage Methods

Prior to HDF Version 3.2, each data element had to be stored in one contiguous block in the basic HDF file. Version 3.2 introduced *extended tags*, a mechanism supporting alternate physical data element storage structures. All NCSA-supported tags with variable-sized data elements can take advantage of the extended tag features.

Extended Tag Implementation

Extended tags are automatically recognized by current versions of the HDF library and interpreted according to a description record. The description record, a complete data element, identifies the type of extended element and provides the relevant parameters for data retrieval.

Extended tags currently support two styles of alternate physical storage:

Linked block elements are stored in several non-contiguous blocks within the basic HDF file.

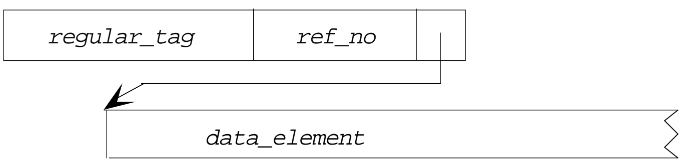
External elements are stored in a separate file, external to the basic HDF file.

Every NCSA-supported tag is represented in HDF libraries and files by a tag number. NCSA-supported tags that take advantage of alternative physical storage features have an alternative tag number, called an *extended tag number*, that appears instead of the original tag number when an alternative physical storage method is in use.

When NCSA determines that an extended tag should be defined for a given tag, the extended tag number is determined by performing an arithmetic OR with the original tag number and the hexadecimal number 0x4000. For example, the tag `DFTAG_RI` points to a data element containing a raster image. If the data element is stored contiguously in the same HDF file, the DD contains the tag number `302`; if the data element is stored either in linked blocks or in an external file, the DD contains the extended tag number `16384`.

If a data object uses a regular tag number, its storage structure will be exactly as described in the "Tag Specifications" section of this chapter. Figure 6.1 illustrates this general structure with the DD pointing directly to a single, contiguous data block.

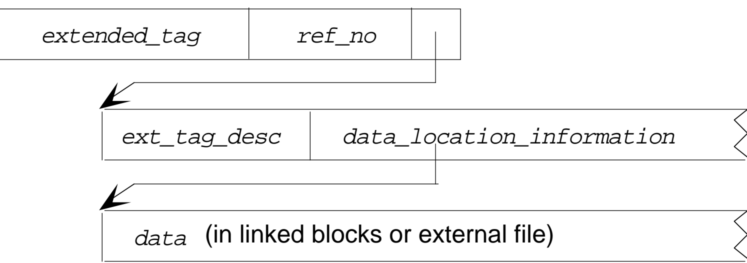
Figure 6.1 Regular Data Object



regular_tag	Tag number
ref_no	Reference number
data_element	The data element

If a data object uses an extended tag, the storage structure will appear generally as illustrated in Figure 6.2. The DD will point to an extended tag description record which in turn will point to the data.

Figure 6.2 Data Object with Extended Tag



extended_tag	Extended tag number
ref_no	Reference number
ext_tag_desc	A 32-bit constant defined in `Hdfi.h` that identifies the type of alternative storage involved. Current definitions include `EXT_LINKED` for linked block elements or `EXT_EXTERN` for external elements.

data_location_information
Information identifying and describing the linked blocks or external file

data — The data, stored either in linked blocks or in an external file

Since the HDF tools were modified for HDF Version 3.2 to handle extended tags automatically, the only thing the user ever has to do is specify the use of either the linked blocks mechanism or an external file. Once that has been specified, the user can forget about extended tags entirely; the HDF library will manage everything correctly.

There is only one circumstance under which an HDF user will need to be concerned with the difference between regular tag numbers and extended tag numbers. If a user bypasses the regular HDF interface to examine a raw HDF file, that user will have to know the extended tag numbers, their significance, and the alternative storage structures.

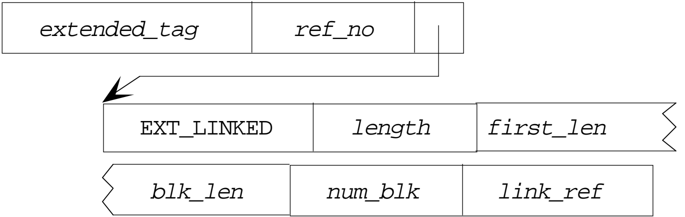
Linked Block Elements

As mentioned above, data elements had to be stored as single contiguous blocks within the basic HDF file prior to HDF Version 3.2. This meant that if a data element grew larger than the allotted space, the file had to be erased from its current location and rewritten at the end of the file.

Linked blocks provide a convenient means of addressing this problem by linking new data blocks to a pre-existing data element. Linked block elements consist of a series of data blocks chained together in a linked list (similar to the DD list). The data blocks must be of uniform size, except for the first block, which is considered a special case.

The linked block data element is a description record beginning with the constant `EXT_LINKED`, which identifies the linked block storage method. The rest of the record describes the organization of the data element stored as linked blocks. Figure 6.3 illustrates a linked block description record.

Figure 6.3 Linked Block Description Record

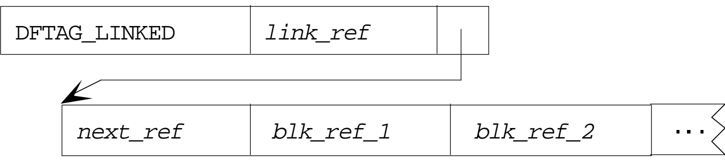


extended_tag	The extended tag counterpart of any NCSA standard tag (16-bit integer)
ref_no	Reference number (16-bit integer)
`EXT_LINKED`	Constant identifying this as a linked block description record (32-bit integer)
length	Length of entire element (32-bit integer)
first_len	Length of the first data block (32-bit integer)
blk_len	Length of successive data blocks (32-bit integer)
num_blk	Number of blocks per block table (32-bit integer)

link_ref Reference number of first block table (16-bit integer)

The *link_ref* field of the description record gives the reference number of the first linked block table for the element. This table is identified by the tag/ref DFTAG_LINKED/*link_ref* and contains *num_blk* entries. There may be any number of linked block tables chained together to describe a linked block element. Figure 6.4 illustrates a linked block table.

Figure 6.4 A Linked Block Table



link_ref Reference number for this table (16-bit integer)

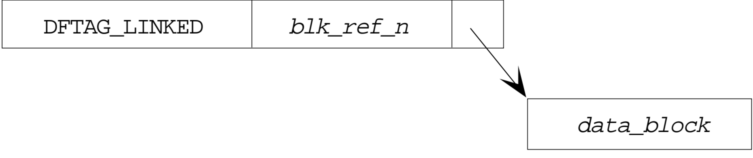
next_ref Reference number for next table (16-bit integer)

blk_ref_n Reference number for data block (16-bit integer)

The *next_ref* field contains the reference number of the next linked block table. A value of zero (0) in this field indicates that there are no additional linked block tables associated with this element.

The *blk_ref_n* fields of each linked block table contain reference numbers for the individual data blocks that make up the data portion of the linked block element. These data blocks are identified by the tag/ref DFTAG_LINKED/*blk_ref_n* as illustrated in Figure 6.5. Although it may seem ambiguous to use the same tag to refer to two different objects, this ambiguity is resolved by the context in which the tags appear.

Figure 6.5 A Data Block



blk_ref_n Reference number for this data block (16-bit integer)

data_block Block of actual data (size specified by *first_len* or *blk_len* in the description record)

Linked block elements can be created using the function `HLcreate()`, which is discussed in Chapter 3, "The HDF General Purpose Interface."

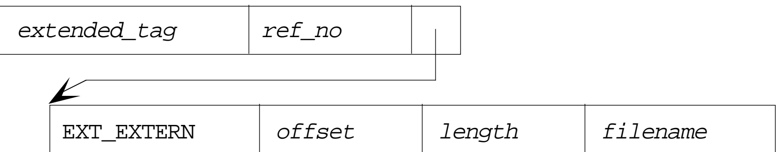
External Elements

External elements allow the data portion of an HDF element to reside in a separate file. The potential of external data elements is largely unexplored in the HDF context, although other file formats (most notably the Common Data Format, CDF, from NASA) have used external data elements to great advantage.

Because there has been little discussion of external elements within the HDF user community, the structure of these elements is still not

completely defined. Figure 6.6 shows a diagram of the suggested structure for an external element.

Figure 6.6 External Element Description Record



extended_tag	The extended tag counterpart of any NCSA standard tag (16-bit integer)
ref_no	Reference number (16-bit integer)
EXT_EXTERN	Constant identifying this as an external element description record (16-bit integer)
offset	Location of the data within the external file (32-bit integer)
length	Length in bytes of the data in the external file (32-bit integer)
filename	Non-null terminated ASCII string naming the external file (any length)

An external element description record begins with the constant EXT_EXTERN, which identifies the data object as having an externally stored data element. The rest of the description record consists of the specific information required to retrieve the data.

External elements can be created using the function HXcreate(), which is discussed in Chapter 3, “The HDF General Purpose Interface.”

Tag Specifications

The following pages contain the specifications of all the NCSA-supported tags in HDF Version 3.3. Each entry contains the following information:

- The tag (in capital letters in the left margin)
- The full name of the tag (on the first line to the right)
- The type and, where possible, the amount of data in the corresponding data element (on the second line to the right)

  When the data element is a variable-sized data structure—such as text, a string, or a variable-sized array—the amount of data cannot be specified exactly. Where possible, a formula is provided to estimate the amount of data. The string `? bytes` appears when neither the size nor the structure of the data element can be specified.
- The tag number in decimal/(hexadecimal) (on the third line to the right)
- A diagram illustrating the structure of the tag and its associated data

  Since all DDs that point to a data element contain data length and data offset fields, these fields are not included in the illustrations.
- A full specification of the tag, including a description of the data element and a discussion of its intended use.

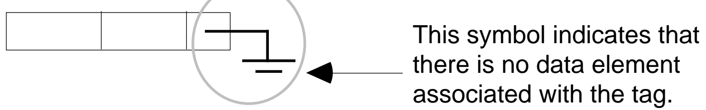
Tags are roughly grouped according to the roles they play:

- Utility tags
- Annotation tags
- Compression tags
- Raster Image tags
- Composite image tags
- Vector image tags
- Scientific data set tags
- Vset tags
- Obsolete tags

These groupings imply a general context for the use of each tag; they are not meant to restrict their use.

Please note the subsection "Obsolete Tags." These tags have fallen out of use with the continuing development of HDF. They are still recognized by the HDF library, but users should not write new objects using them; they may eventually be dropped from the HDF specification.

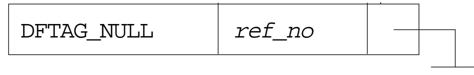
In the following discussion, the ground symbol indicates that the DD for this tag includes no pointer to a data element. I.e., there is never a data element associated with the tag.



Utility Tags

DFTAG_NULL

No data
0 bytes
1 (0x0001)

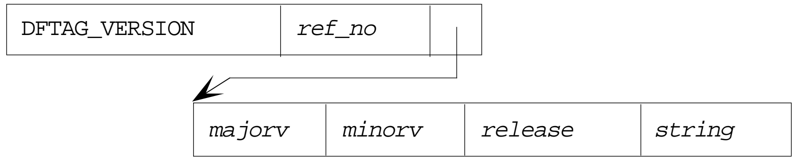


ref_no	Reference number (16-bit integer; always 0)

This tag is used for place holding and to fill empty portions of the data description block. The length and offset fields (not shown) of a DFTAG_NULL DD must be zero (0).

DFTAG_VERSION

Library version number
12 bytes plus the length of a string
30 (0x001E)

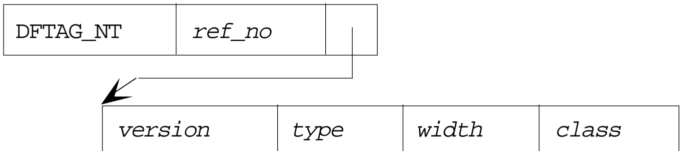


ref_no	Reference number (16-bit integer)
majorv	Major version number (32-bit integer)
minorv	Minor version number (32-bit integer)
release	Release number (32-bit integer)
string	Non-null terminated ASCII string (any length)

The data portion of this tag contains the complete version number and a descriptive string for the latest version of the HDF library to write to the file.

DFTAG_NT

Number type
4 bytes
106 (0x006A)



ref_no	Reference number (16-bit integer)
version	Version number of NT information (8-bit integer)
type	Unsigned integer, signed integer, unsigned character, character, floating point, double precision floating point (8-bit code)
width	Number of bits, all of which are assumed to be significant (8-bit code)
class	A generic value, with different interpretations depending on type: floating point, integer, or character (8-bit code)

Several values that may be used for each of the three types in the field CLASS are listed in Table 6.1. This is not an exhaustive list.

Table 6.1 Number Type Values

Type	Mnemonic	Value
Floating point	DFNTF_NONE	0
	DFNTF_IEEE	1
	DFNTF_VAX	2
	DFNTF_CRAY	3
	DFNTF_PC	4
	DFNTF_CONVEX	5
Integer	DFNTI_MBO	1
	DFNTI_IBO	2
	DFNTI_VBO	4
Character	DFNTC_ASCII	1
	DFNTC_EBCDOC	2
	DFNTC_BYTE	0

The number type flag is used by any other element in the file to indicate specifically what a numeric value looks like. Other tag types should contain a reference number pointer to an DFTAG_NT instead of containing their own number type definitions.

The version field allows expansion of the number type information, in case some future number types cannot be described using the fields currently defined. Successive versions of the DFTAG_NT may be substantially different from the current definition, but backward compatibility will be maintained. The current DFTAG_NT version number is 1.

DFTAG_MT

Machine type
0 bytes
107 (0x006B)

DFTAG_MT	*double*	*float*	*int*	*char*	

double Specifies method of encoding double precision floating point (4-bit code)

float Specifies method of encoding single precision floating point (4-bit code)

int Specifies method of encoding integers (4-bit code)

char Specifies method of encoding characters (4-bit code)

DFTAG_MT specifies that all unconstrained or partially constrained values in this HDF file are of the default type for that hardware. When DFTAG_MT is set to VAX, for example, all integers will be assumed to be in VAX byte order unless specifically defined otherwise with a DFTAG_NT tag. Note that all of the headers and many tags, the whole raster image set for example, are defined with bit-wise precision and will not be overridden by the DFTAG_MT setting.

For DFTAG_MT, the reference field itself is the encoding of the DFTAG_MT information. The reference field is 16 bits, taken as four groups of four bits, specifying the types for double-precision floating point, floating point, integer, and character respectively. This allows 16 generic specifications for each type.

To the user, these will be defined constants in the header file hdf.h, specifying the proper descriptive numbers for Sun, VAX, Cray, Convex, and other computer systems. If there is no DFTAG_MT in a file, the application may assume that the data in the file has been written on the local machine; any portability problems must be addressed by the user. For this reason, we recommend that all HDF files contain a DFTAG_MT for maximum portability.

Currently available data encodings are listed in Table 6.2.

Table 6.2 Available Machine Types

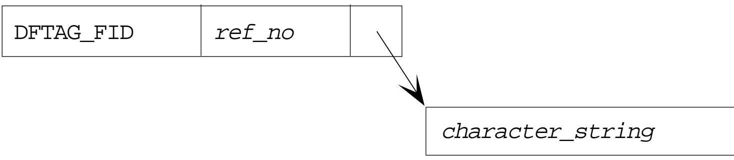
Type	Available Encodings
Double precision floating point	IEEE64 VAX64 CRAY128
Floating point	IEEE32 VAX32 CRAY64
Integers	VAX32 Intel16 Intel32 Motorola32 CRAY64
Characters	ASCII EBCDIC

New encodings can be added for each data type as the need arises.

Annotation Tags

`DFTAG_FID`

File identifier
String
100 (0x0064)

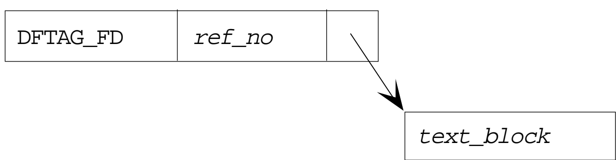


ref_no	Reference number (16-bit integer)
character_string	Non-null terminated ASCII text (any length)

This tag points to a string which the user wants to associate with this file. The string is not null terminated. The string is intended to be a user-supplied title for the file.

`DFTAG_FD`

File description
Text
101 (0x0065)

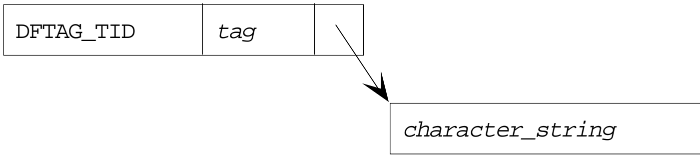


ref_no	Reference number (16-bit integer)
text_block	Non-null terminated ASCII text (any length)

This tag points to a block of text describing the overall file contents. The text can be any length. The block is not null terminated. The text is intended to be user-supplied comments about the file.

`DFTAG_TID`

Tag identifier
String
102 (0x0066)



tag Tag number to which this tag refers (16-bit integer)

character_string
Non-null terminated ASCII text (any length)

The data for this tag is a string that identifies the functionality of the tag indicated in the space normally used for the reference number. For

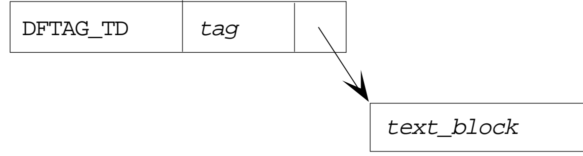
example, the tag identifier for DFTAG_TID might point to data that reads "tag identifier."

Many tags are identified in the HDF specification, so it is usually unnecessary to include their identifiers in the HDF file. But with user-defined tags or special-purpose tags, the only way for a human reader to diagnose what kind of data is stored in a file is to read tag identifiers. Use tag descriptions to define even more detail about your user-defined tags.

Note that with this tag you may make use of the user-defined tags to check for consistency. Although two persons may use the same user-defined tag, they probably will not use the same tag identifier.

DFTAG_TD

Tag description
Text
103 (0x0067)



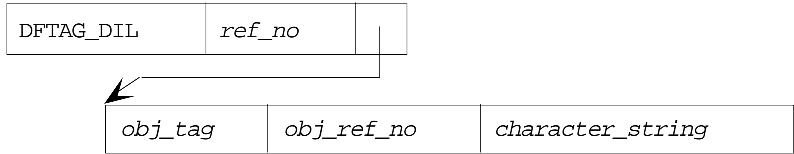
tag	Tag number to which this tag refers (16-bit integer)
text_block	Non-null terminated ASCII text (any length)

The data for this tag is a text block which describes in relative detail the functionality and format of the tag which is indicated in the space normally occupied by the reference number. This tag is intended to be used with user-defined tags and provides a medium for users to exchange files that include human-readable descriptions of the data.

It is important to provide everything that a programmer might need to know to read the data from your user-defined tag. At the minimum, you should specify everything you would need to know in order to retrieve your data at a later date if the original program were lost.

DFTAG_DIL

Data identifier label
String
104 (0x0068)



ref_no Reference number (16-bit integer)

obj_tag Tag number of the data to which this label applies (16-bit integer)

obj_ref_no Reference number of the data object to which this label applies (16-bit integer)

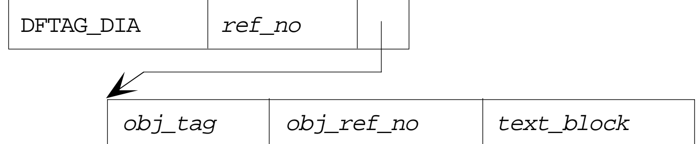
character_string Non-null terminated ASCII text (any length)

The DFTAG_DIL data object consists of a tag/ref followed by a string. The string serves as a label for the data identified by the tag/ref.

By including DFTAG_DIL tags, you can give a data object a label for future reference. For example, DFTAG_DIL can be used to assign titles to images.

DFTAG_DIA

Data identifier annotation
Text
105 (0x0069)



ref_no Reference number (16-bit integer)

obj_tag Tag number of the data to which this annotation applies (16-bit integer)

obj_ref_no Reference number of the data object to which this annotation applies (16-bit integer)

text_block Non-null terminated ASCII text (any length)

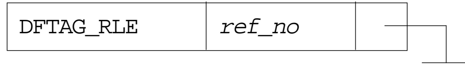
The DFTAG_DIA data object consists of a tag/ref followed by a text block. The text block serves as an annotation of the data identified by the tag/ref.

With a DFTAG_DIA tag, any data object can have a lengthy, user-written description. This can be used to include comments about images, data sets, source code, and so forth.

Compression Tags

DFTAG_RLE

Run length encoded data
0 bytes
11 (0x000B)



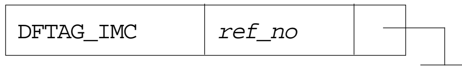
ref_no Reference number (16-bit integer)

This tag is used in the DFTAG_ID compression field and in other places to indicate that an image or section of data is encoded with a run-length encoding scheme. The RLE method used is byte-wise. Each run is preceded by a count byte. The low seven bits of the count byte indicate the number of bytes (n). The high bit of the count byte indicates whether the next byte should be replicated *n* times (high bit = 1), or whether the next *n* bytes should be included as is (high bit = 0).

See also: DFTAG_ID in "Raster Image Tags"
DFTAG_NDG in "Scientific Data Set Tags"

DFTAG_IMC

IMCOMP compressed data
0 bytes
12 (0x000C)



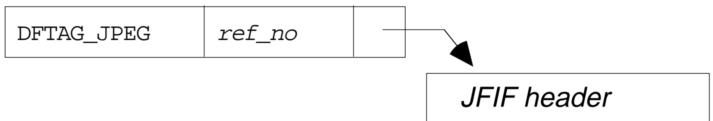
ref_no Reference number (16-bit integer)

This tag is used in the DFTAG_ID compression field and in other places to indicate that an image or section of data is encoded with an IMCOMP encoding scheme. This scheme is a 4:1 aerial averaging method which is easy to decompress. It counts color frequencies in 4x4 squares to optimize color sampling.

See also: DFTAG_ID in "Raster Image Tags"
DFTAG_NDG in "Scientific Data Set Tags"

DFTAG_JPEG

24-bit JPEG compression information
? bytes
13 (0x000D)

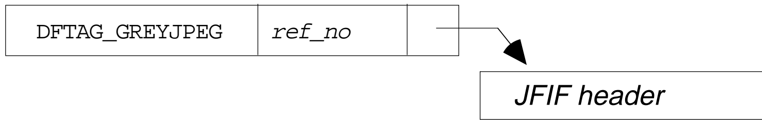


ref_no Reference number (16-bit integer)

This tag points to header information for 24-bit JPEG compressed images. The data in this tag is identical to the header data stored in a JFIF (JPEG File Interchange Format) file up to the start-of-frame parameter. The start-of-frame parameter and all further data for the JPEG image is stored in the associated DFTAG_CI data element which is the companion to the DFTAG_JPEG element. (See the document *JPEG File Interchange Format** for a detailed description of the file format.)

DFTAG_GREYJPEG

8-bit JPEG compression information
? bytes
14 (0x000E)



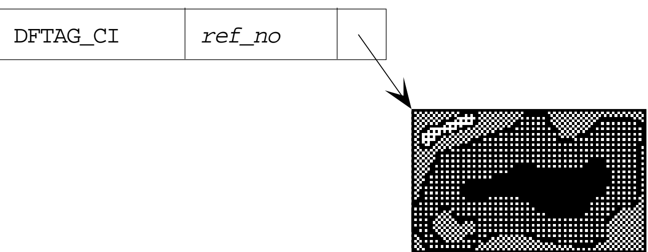
ref_no Reference number (16-bit integer)

This tag points to header information for 8-bit JPEG compressed images. The data in this tag is identical to the header data stored in a JFIF (JPEG File Interchange Format) file up to the start-of-frame parameter (see the JFIF format document for further details). The start-of-frame parameter and all further data for the JPEG image is stored in the associated DFTAG_CI data element which is the companion to the DFTAG_JPEG element.

* The document *JPEG File Interchange Format* has not been published in a regular periodical. An electronic copy is available as a Postscript file from NCSA's FTP server ftp.ncsa.uiuc.edu in the same directory as this document, *NCSA HDF Specification and Developer's Guide*. Printed copies are available from C-Cube Microsystems, 1778 McCarthy Boulevard, Milpitas, CA 95035 (phone: 408-944-6300. Fax: 408-944-6314. Current email contact: eric@c3.pla.ca.us).

DFTAG_CI

Compressed raster image
? bytes
303 (0x012F



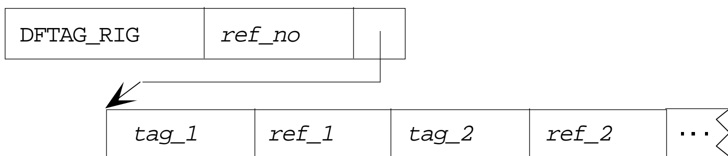
ref_no Reference number (16-bit integer)

This tag points to a stream of bytes that make up a compressed image. The type of compression, together with any necessary parameters, are stored as a separate data object. For example, if DFTAG_JPEG is contained in the same raster image group, the stream of bytes contains the sratt-of-frame parameter and all further data for the JPEG image. Other parameters are stored in the DFTAG_JPEG object.

Raster Image Tags

DFTAG_RIG

Raster image group
*n**4 bytes (where *n* is the number of data objects in the group)
306 (0x0132)



ref_no	Reference number (16-bit integer)
tag_n	Tag number for n^{th} member of the group (16-bit integer)
ref_n	Reference number for n^{th} member of the group (16-bit integer)

The RIG data element contains the tag/refs of all the data objects required to display a raster image correctly. Application programs that deal with RIGs should read all the elements of a RIG and process those identifiers which it can display correctly. Even if the application cannot process *all* of the objects, the objects that it can process will be usable.

Table 6.3 lists the tags that may appear in an RIG.

Table 6.3 Available RIG Tags

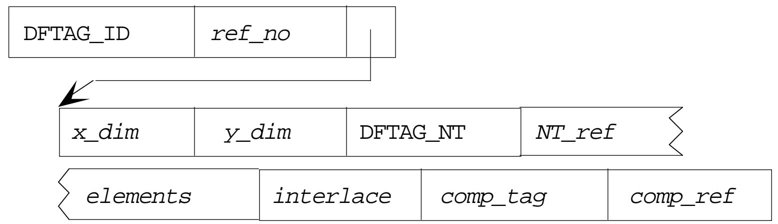
Tag	Description
DFTAG_ID	Image dimension record
DFTAG_RI	Raster image
DFTAG_XYP	X-Y position
DFTAG_LD	LUT dimension
DFTAG_LUT	Color lookup table
DFTAG_MD	Matte channel dimension
DFTAG_MA	Matte channel
DFTAG_CCN	Color correction
DFTAG_CFM	Color format
DFTAG_AR	Aspect ratio

Example

DFTAG_ID, DFTAG_RI, DFTAG_LD, DFTAG_LUT
Assume that an image dimension record, a raster image, an LUT dimension record, and an LUT are all required to display a particular raster image correctly. These data objects can be associated in an RIG so that an application can read the image dimensions then the image. It will then read the lookup table and display the image.

DFTAG_ID
DFTAG_LD
DFTAG_MD

DFTAG_ID	DFTAG_LD	DFTAG_MD
Image dimension	LUT dimension	Matte dimension
20 bytes	20 bytes	20 bytes
300 (0x012C)	307 (0x0133)	308 (0x0134)



ref_no	Reference number (16-bit integer)
x_dim	Length of x (horizontal) dimension (32-bit integer)
y_dim	Length of y (vertical) dimension (32-bit integer)
NT_ref	Reference number for number type information
elements	Number of elements that make up one entry (16-bit integer)
interlace	Type of interlacing used (16-bit integer) 0 The components of each pixel are together. 1 Color elements are grouped by scan lines. 2 Color elements are grouped by planes.
comp_tag	Tag which tells the type of compression used and any associated parameters (16-bit integer)
comp_ref	Reference number of compression tag (16-bit integer)

These three dimension records have exactly the same format; they specify the dimensions of the 2-dimensional arrays after which they are named and provide information regarding other attributes of the data in the array:

    DFTAG_ID specifies the dimensions of a DFTAG_RI.
    DFTAG_LD specifies the dimensions of a DFTAG_LUT.
    DFTAG_MD specifies the dimensions of a DFTAG_MA.

Other attributes described in the image dimension record include the number type of the elements, the number of elements per pixel, the interlace scheme used, and the compression scheme used (if any).

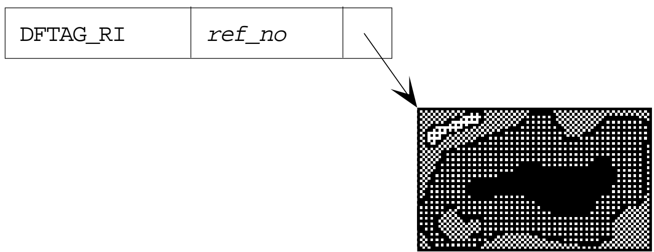
For example, a 512x256 row-wise 24-bit raster image with each pixel stored as RGB bytes would have the following values:

x_dim	512
y_dim	256
NT_ref	UINT8
elements	3 (3 elements per pixel: e.g., R, G, and B)
interlace	0 (RGB values not separated)
comp_tag	0 (no compression is used)

The diagram above illustrates the tag DFTAG_ID. The DFTAG_LD and DFTAG_MD diagrams would be identical except for the tag name in the fist cell, whch would be DFTAG_LD and DFTAG_MD, respectively.

`DFTAG_RI`

Raster image
*xdim*ydim*elements*NTsize* bytes (*xdim*, *ydim*, *elements,* and *NTsize* are specified in the corresponding `DFTAG_ID`)
302 (0x012E)

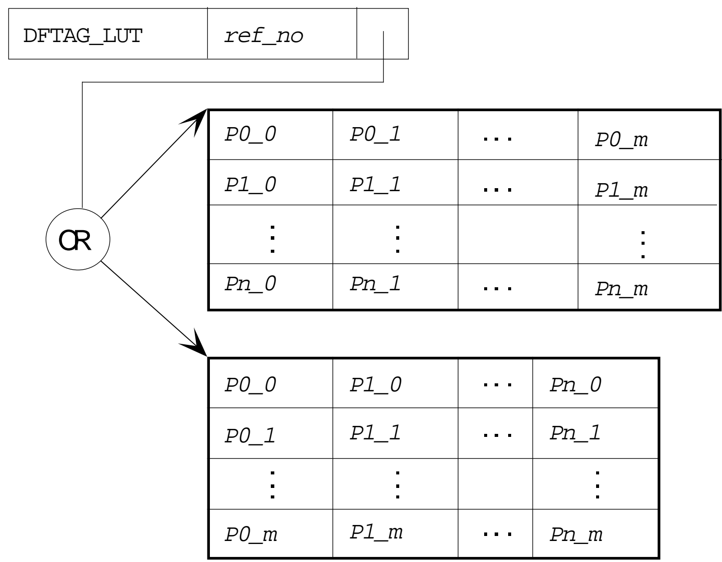


ref_no	Reference number (16-bit integer)

This tag points to raster image data. It is stored in row-major order and must be interpreted as specified by *interlace* in the related `DFTAG_ID`.

`DFTAG_LUT`

Lookup table
*xdim*ydim*elements*NTsize* bytes (*xdim*, *ydim*, *elements,* and *NTsize* are specified in the corresponding `DFTAG_ID`)
301 (0x012D)



ref_no	Reference number (16-bit integer)
Pn_m	m[th] value of parameter n (size is specified by the `DFTAG_NT` in the corresponding `DFTAG_LD`)

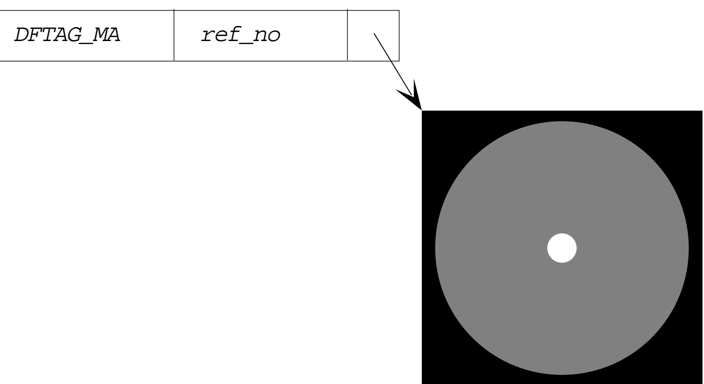
The `DFTAG_LUT`, sometimes called a palette, is used to assign colors to data values. When a raster image consists of data values which are going to be interpreted through an LUT capability, the `DFTAG_LUT` should be loaded along with the image.

The most common lookup table is the RGB lookup table which will have X dimension = 256 and Y dimension = 1 with three elements per

entry, one each for red, green, and blue. The interlace will be either 0, where the LUT values are given RGB, RGB, RGB, ..., or 1, where the LUT values are given as 256 reds, 256 greens, 256 blues.

DFTAG_MA

Matte channel
*xdim*ydim*elements*NTsize* bytes (*xdim*, *ydim*, *elements,* and *NTsize* are specified in the corresponding DFTAG_ID)
309 (0x0135)

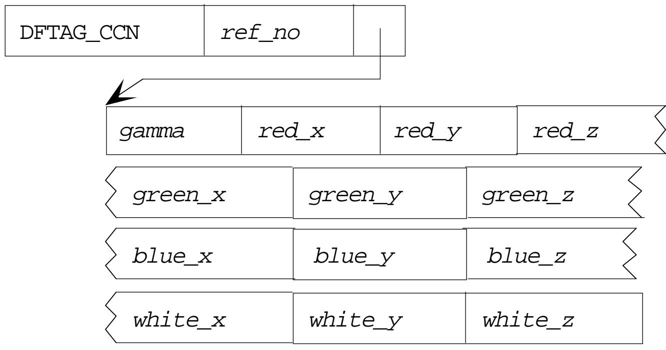


ref_no Reference number (16-bit integer)

The DFTAG_MA data object contains transparency data which can be used to facilitate the overlaying of images. The data consists of a 2-dimensional array of unsigned 8-bit integers ranging from 0 to 255. Each point in a DFTAG_MA indicates the transparency of the corresponding point in a raster image of the same dimensions. A value of 0 indicates that the data at that point is to be considered totally transparent, while a value of 255 indicates that the data at that point is totally opaque. It is assumed that a linear scale applies to the transparency values, but users may opt to interpret the data in any way they wish.

DFTAG_CCN

Color correction
52 bytes (usually)
310 (0x0136)



ref_no Reference number (16-bit integer)

gamma Gamma parameter (32-bit IEEE floating point)

red_x, *red_y*, and *red_z*
Red x, y, and z correction factors (32-bit IEEE floating point)

green_x, *green_y*, and *green_z*
Green x, y, and z correction factors (32-bit IEEE floating point)

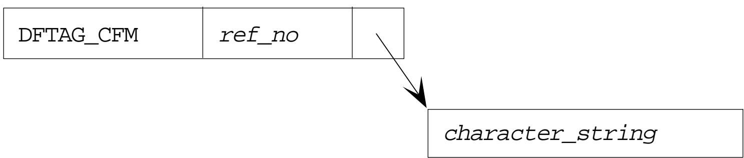
blue_x, *blue_y*, and *blue_z*
Blue x, y, and z correction factors (32-bit IEEE floating point)

white_x, *white_y*, and *white_z*
White x, y, and z correction factors (32-bit IEEE floating point)

Color correction specifies the Gamma correction for the image and color primaries for the generation of the image.

DFTAG_CFM

Color format
String
311 (0x0137)



ref_no Reference number (16-bit integer)

character_string Non-null terminated ASCII string (any length)

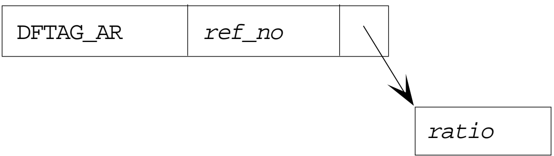
The color format data element contains a string of uppercase characters that indicates how each element of each pixel in a raster image is to be interpreted. Table 6.4 lists the available color format strings.

Table 6.4 Color Format String Values

String	Description
VALUE	Pseudo-color, or just a value associated with the pixel
RGB	Red, green, blue model
XYZ	Color-space model
HSV	Hue, saturation, value model
HSI	Hue, saturation, intensity
SPECTRAL	Spectral sampling method

DFTAG_AR

Aspect ratio
4 bytes
312 (0x0138)



ref_no Reference number (16-bit integer)

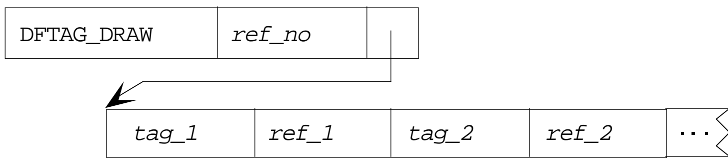
ratio Ratio of width to height (32-bit IEEE float)

The data for this tag is the visual aspect ratio for this image. The image should be visually correct if displayed on a screen with this aspect ratio. The data consists of one floating-point number which represents width divided by height. An aspect ratio of 1.0 indicates a display with perfectly square pixels; 1.33 is a standard aspect ratio used by many monitors.

Composite Image Tags

DFTAG_DRAW

Draw
*n**4 bytes (where *n* is the number of data objects that make up the composite image)
400 (0x0190)



ref_no Reference number (16-bit integer)

tag_n Tag number of the nth member of the draw list (16-bit integer)

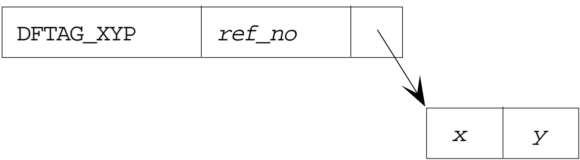
ref_n Reference number of the nth member of the draw list (16-bit integer)

The DFTAG_DRAW data element consists of a list of tag/refs that define a composite image. The data objects indicated should be displayed in order. This can include several RIGs which are to be displayed simultaneously. It can also include vector overlays, like DFTAG_T14, which are to be placed on top of an RIG.

Some of the elements in a DFTAG_DRAW list may be instructions about how images are to be composited (XOR, source put, anti-aliasing, etc.). These are defined as individual tags.

`DFTAG_XYP`

XY position
8 bytes
500 (0x01F4)



ref_no	Reference number (16-bit integer)
x	X-coordinate (32-bit integer)
y	Y-coordinate (32-bit integer)

`DFTAG_XYP` is used in composites and other groups to indicate an XY position on the screen. For this, (0,0) is the lower left corner of the print area. X is the number of pixels to the right along the horizontal axis and Y is the number of pixels up on the vertical axis. The X and Y coordinates are two 32-bit integers.

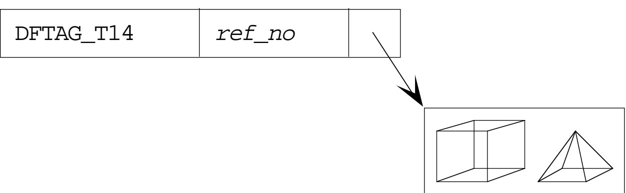
For example, if `DFTAG_XYP` is present in a `DFTAG_RIG`, the `DFTAG_XYP` specifies the position of the lower left corner of the raster image on the screen.

See also: `DFTAG_DRAW` in this section

Vector Image Tags

`DFTAG_T14`

Tektronix 4014
? bytes
602 (0x25A)

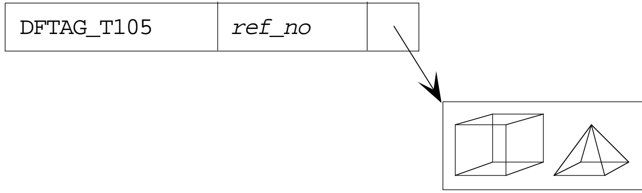


ref_no Reference number (16-bit integer)

This tag points to a Tektronix 4014 data stream. The bytes in the data field, when read and sent to a Tektronix 4014 terminal, will display a vector image. Only the lower seven bits of each byte are significant. There are no record markings or non-Tektronix codes in the data.

`DFTAG_T105`

Tektronix 4105
? bytes
603 (0x25B)



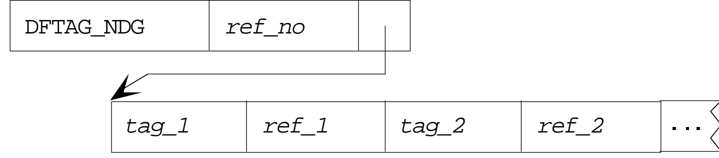
ref_no Reference number (16-bit integer)

This tag points to a Tektronix 4105 data stream. The bytes in the data field, when read and sent to a Tektronix 4105 terminal, will be displayed as a vector image. Only the lower seven bits of each byte are significant. Some terminal emulators will not correctly interpret every feature of the Tektronix 4105 terminal, so you may wish to use only a subset of the available Tektronix 4105 vector commands.

Scientific Data Set Tags

DFTAG_NDG

Numeric data group
*n**4 bytes (where *n* is the number of data objects in the group.)
720 (0x02D0)



ref_no Reference number (16-bit integer)

tag_n Tag number of nth member of the group (16-bit integer)

ref_n Reference number of nth member of the group (16-bit integer)

The NDG data contains a list of tag/refs that define a scientific data set. DFTAG_NDG supersedes the old DFTAG_SDG, which became obsolete upon the release on HDF Version 3.2. A more complete explanation of the relationship between DFTAG_NDG and DFTAG_SDG can be found in Chapter 4, "Sets and Groups."

All of the members of an NDG provide information for correctly interpreting and displaying the data. Application programs that deal with NDGs should read all of the elements of a NDG and process those data objects which it can use. Even if an application cannot process all of the objects, the objects that it can understand will be usable.

Table 6.5 lists the tags that may appear in an NDG.

Table 6.5 Available NDG Tags

Tag	Description
DFTAG_SDD	Scientific data dimension record (rank and dimensions)
DFTAG_SD	Scientific data
DFTAG_SDS	Scales
DFTAG_SDL	Labels
DFTAG_SDU	Units
DFTAG_SDF	Formats
DFTAG_SDM	Maximum and minimum values
DFTAG_SDC	Coordinate system
DFTAG_CAL	Calibration information
DFTAG_FV	Fill value
DFTAG_LUT	Color lookup table
DFTAG_LD	Lookup table dimension record
DFTAG_SDLNK	Link to old-style DFTAG_SDG

Example

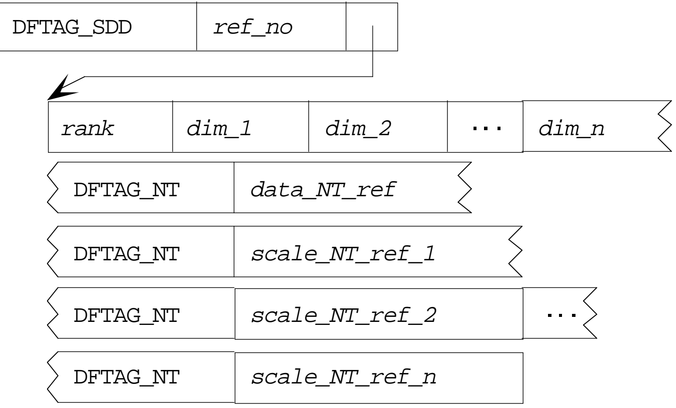
DFTAG_SDD, DFTAG_SD, DFTAG_SDM

Suppose that an NDG contains a dimension record, scientific data, and the maximum and minimum values of the data. These data objects can be associated in an NDG so that an application can read the rank and dimensions from the dimension record and then read the data array. If the application needs maximum and minimum values, it will read them as well.

See also: Chapter 4, “Sets and Groups”

DFTAG_SDD

Scientific data dimension record
6 + 8**rank* bytes
701 (0x02BD)



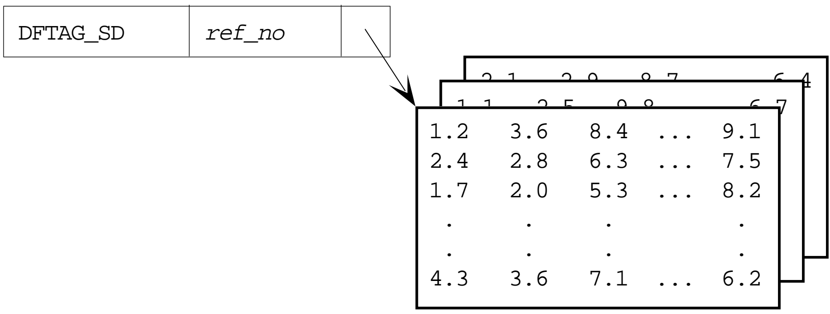
ref_no	Reference number (16-bit integer)
rank	Number of dimensions (16-bit integer)
dim_n	Number of values along the nth dimension (32-bit integer)
data_NT_ref	Reference number of DFTAG_NT for data (16-bit integer)
scale_NT_ref_n	Reference number for DFTAG_NT for the scale for the nth dimension (16-bit integer)

This record defines the rank and dimensions of the array in the scientific data set. For example, a DFTAG_SDD for a 500x600x3 array of floating-point numbers would have the following values and components.

Rank: 3
Dimensions: 500, 600, and 3.
One data NT
Three scale NTs

`DFTAG_SD`

Scientific data
*NTsize*x*y*z**... bytes (where NTsize is the size of the data NT specified in the corresponding `DFTAG_SDD` and *x, y, z,* etc. are the dimension sizes)
702 (0x02BE)

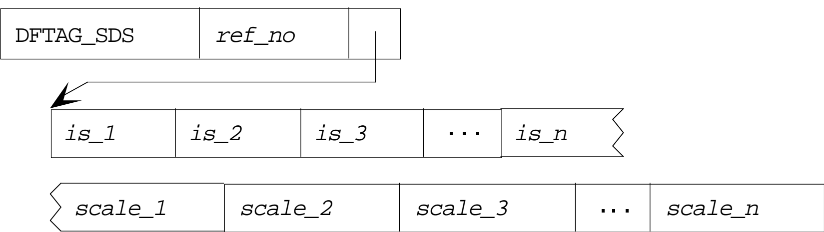


ref_no	Reference number (16-bit integer)

This tag points to an array of scientific data. The type of the data may be specified by an `DFTAG_NT` included with the SDG. If there is no `DFTAG_NT`, the type of the data is floating-point in standard IEEE 32-bit format. The rank and dimensions must be stored as specified in the corresponding `DFTAG_SDD`. The diagram above shows a 3-dimensional data array.

`DFTAG_SDS`

Scientific data scales
*rank + NTsize0*x + NTsize1*y +NTsize2*z +*... bytes (where *rank* is the number of dimensions, *x*, *y*, *z*, etc. are the dimension sizes, and *NTsize#* are the sizes of each scale NT from the corresponding `DFTAG_SDD`)
703 (0x02BF)

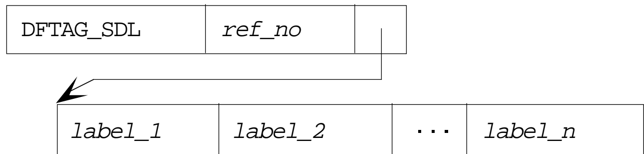


ref_no	Reference number (16-bit integer)
is_n	A flag indicating whether a scale exists for the nth dimension (8-bit integer; `0` or `1`)
scale_n	List of scale values for the nth dimension (type specified in corresponding `DFTAG_SDD`)

This tag points to the scales for the data set. The first *n* bytes indicate whether there is a scale for the corresponding dimension (`1` = yes, `0` = no). This is followed by the scale values for each dimension. The scale consists of a simple series of values where the number of values and their types are specified in the corresponding `DFTAG_SDD`.

DFTAG_SDL

Scientific data labels
? bytes
704 (0x02C0)



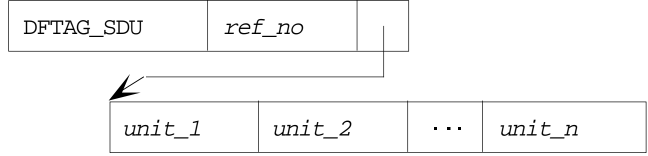
ref_no Reference number (16-bit integer)

label_n Null terminated ASCII string (any length)

This tag points to a list of labels for the data in each dimension of the data set. Each label is a string terminated by a null byte (0).

DFTAG_SDU

Scientific data units
? bytes
705 (0x02C1)



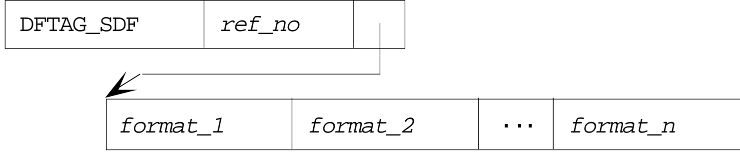
ref_no Reference number (16-bit integer)

unit_n Null terminated ASCII string (any length)

This tag points to a list of strings specifying the units for the data and each dimension of the data set. Each unit's string is terminated by a null byte (0).

DFTAG_SDF

Scientific data format
? bytes
706 (0x02C2)



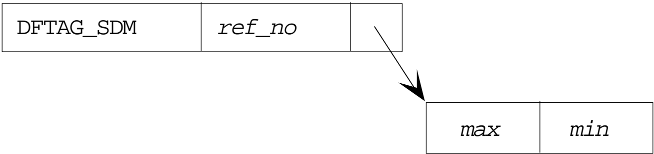
ref_no Reference number (16-bit integer)

format_n Null terminated ASCII string (any length)

This tag points to a list of strings specifying an output format for the data and each dimension of the data set. Each format string is terminated by a null byte (0).

`DFTAG_SDM`

Scientific data max/min
8 bytes
707 (0x02C3)

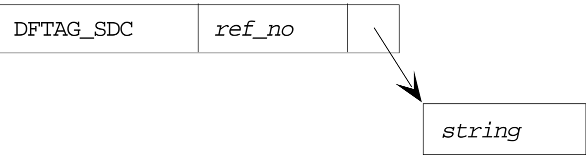


ref_no	Reference number (16-bit integer)
max	Maximum value (type is specified by the data NT in the corresponding `DFTAG_SDD`)
min	Minimum value (type is specified by the data NT in the corresponding `DFTAG_SDD`)

This record contains the maximum and minimum data values in the data set. The type of *max* and *min* are specified by the data NT of the corresponding `DFTAG_SDD`.

`DFTAG_SDC`

Scientific data coordinates
? bytes
708 (0x02C4)

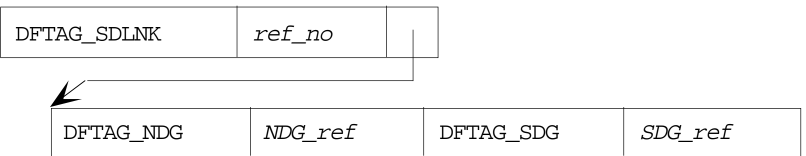


ref_no	Reference number (16-bit integer)
string	Null terminated ASCII string (any length)

This tag points to a string specifying the coordinate system for the data set. The string is terminated by a null byte.

DFTAG_SDLNK

Scientific data set link
8 bytes
710 (0x02C6)



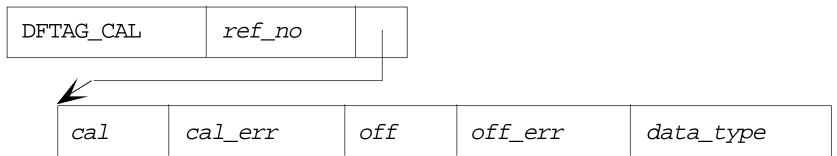
ref_no	Reference number (16-bit integer)
DFTAG_NDG	NDG tag (16-bit integer)
NDG_ref	NDG reference number (16-bit integer)
DFTAG_SDG	SDG tag (16-bit integer)
SDG_ref	SDG reference number (16-bit integer)

The purpose of this tag is to link together an old-style DFTAG_SDG and a DFTAG_NDG in cases where the NDG contains 32-bit floating point data and is, therefore, equivalent to an old SDG.

See also: Chapter 4, “Sets and Groups”

DFTAG_CAL

Calibration information
36 bytes
731 (0x02DB)



ref_no	Reference number (16-bit integer)
cal	Calibration factor (64-bit IEEE float)
cal_err	Error in calibration factor (64-bit IEEE float)
off	Calibration offset (64-bit IEEE float)
off_err	Error in calibration offset (64-bit IEEE float)
data_type	Constant representing the effective data type of the calibrated data (32-bit integer)

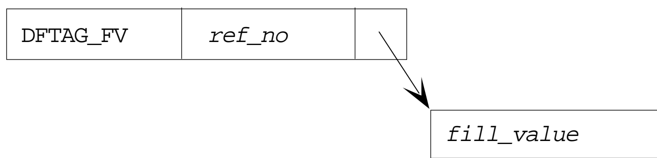
This tag points to a calibration record for the associated DFTAG_SD. The data can be calibrated by first multiplying by the *cal* factor, then adding the *off* value. Also included in the record are errors for the calibration factor and offset and a constant indicating the effective data type of the calibrated data. Table 6.6 lists the available *data_type* values.

Table 6.6 Available Calibrated Data Types

Data Type	Description
DFTNT_INT8	Signed 8-bit integer
DFTNT_UINT8	Unsigned 8-bit integer
DFTNT_INT16	Signed 16-bit integer
DFTNT_UINT16	Unsigned 16-bit integer
DFTNT_INT32	Signed 32-bit integer
DFTNT_UINT32	Unsigned 32-bit integer
DFTNT_FLOAT32	32-bit floating point
DFTNT_FLOAT64	64-bit floating point

DFTAG_FV

Fill value
? bytes (size determined by size of data NT in corresponding DFTAG_SDD)
732 (0x02DC)



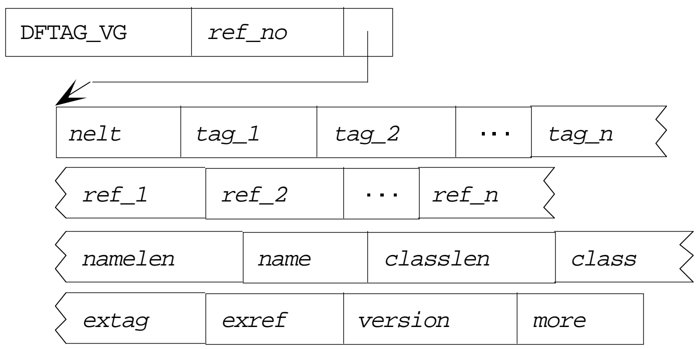
ref_no	Reference number (16-bit integer)
fill_value	Value representing unset data in the corresponding DFTAG_SD (size determined by size of data NT in corresponding DFTAG_SDD)

This tag points to a value which has been used to indicate unset values in the associated DFTAG_SD. The number type of the value (and, therefore, its size) is given in the corresponding DFTAG_SDD.

Vset Tags

DFTAG_VG

Vgroup
14 + 4**nelt* + *namelen* + *classlen* bytes
1965 (0x07AD)

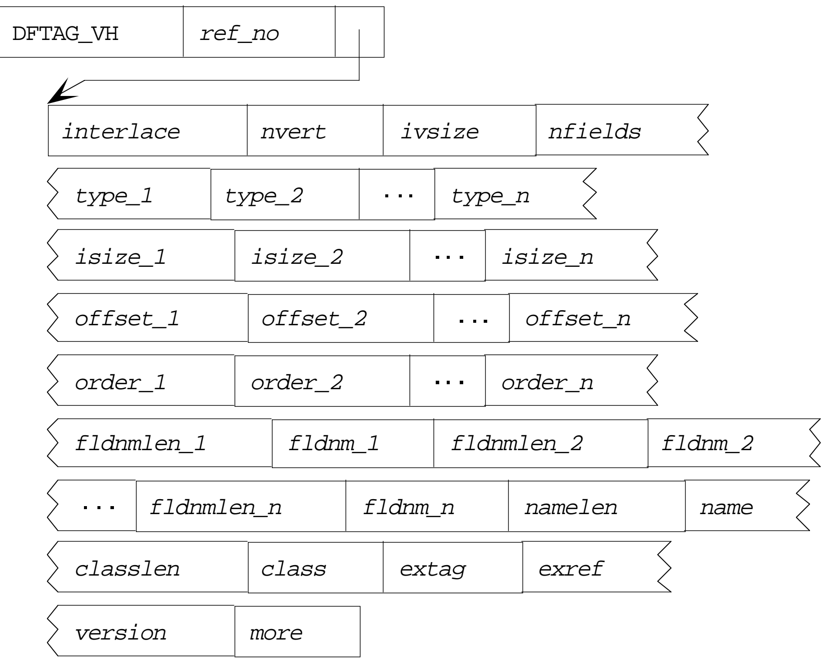


ref_no	Reference number (16-bit integer)
nelt	Number of elements in the Vgroup (16-bit integer)
tag_n	Tag of the nth member of the Vgroup (16-bit integer)
ref_n	Reference number of the nth member of the Vgroup (16-bit integer)
namelen	Length of the name field (16-bit integer)
name	Non-null terminated ASCII string (length given by *namelen*)
classlen	Length of the class field (16-bit integer)
class	Non-null terminated ASCII string (length given by *classlen*)
extag	Extension tag (16-bit integer)
exref	Extension reference number (16-bit integer)
version	Version number of DFTAG_VG information (16-bit integer)
more	Unused (2 zero bytes)

DFTAG_VG provides a general-purpose grouping structure which can be used to impose a hierarchical structure on the tags in the group. Any HDF tag may be incorporated into a Vgroup, including other DFTAG_VG tags.

See also: "Vsets, Vdatas, and Vgroups" in Chapter 4, "Sets and Groups"
NCSA HDF Vsets, Version 2.0 for HDF Version 3.2 and earlier
NCSA HDF User's Guide and *NCSA HDF Reference Manual* for HDF Version 3.3

`DFTAG_VH`	Vdata description
	22 + 10**nfields* + S*fldnmlen n* + *namelen* + *classlen* bytes
	1962 (0x07AA)



ref_no	Reference number (16-bit integer)
interlace	Constant indicating interlace scheme used (16-bit integer)
nvert	Number of entries in Vdata (32-bit integer)
ivsize	Size of one Vdata entry (16-bit integer)
nfields	Number of fields per entry in the Vdata (16-bit integer)
type_n	Constant indicating the data type of the nth field of the Vdata (16-bit integer)
isize_n	Size in bytes of the nth field of the Vdata (16-bit integer)
offset_n	Offset of the nth field within the Vdata (16-bit integer)
order_n	Order of the nth field of the Vdata (16-bit integer)
fldnmlen_n	Length of the nth field name string (16-bit integer)
fldnm_n	Non-null terminated ASCII string (length given by corresponding *fldnmlen_n*)
namelen	Length of the name field (16-bit integer)
name	Non-null terminated ASCII string (length given by *namelen*)
classlen	Length of the class field (16-bit integer)
class	Non-null terminated ASCII string (length given by *classlen*)
extag	Extension tag (16-bit integer)
exref	Extension reference number (16-bit integer)

version	Version number of DFTAG_VH information (16-bit integer)
more	Unused (2 zero bytes)

DFTAG_VH provides all the information necessary to process a DFTAG_VS.

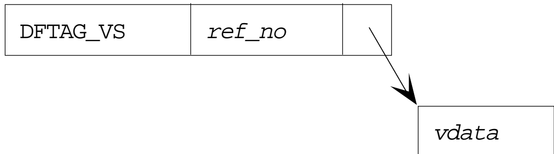
See also: DFTAG_VS (this section)
"Vsets, Vdatas, and Vgroups" in Chapter 4, "Sets and Groups"
NCSA HDF Vsets, Version 2.0 for HDF Version 3.2 and earlier
NCSA HDF User's Guide and *NCSA HDF Reference Manual* for HDF Version 3.3

DFTAG_VS

Vdata

$nvert * \sum_{n=1}^{nfields} (isize_n * order_n)$ bytes, where *nvert, isize_n,* and *order_n* are specified in the corresponding DFTAG_VH

1963 (0x07AB)



ref_no	Reference number (16-bit integer)
vdata	Data block interpreted according to the corresponding DFTAG_VH ($nvert * \sum_{n=1}^{nfields} (isize_n * order_n)$ bytes, where *nvert, isize_n,* and *order_n* are specified in the corresponding DFTAG_VH)

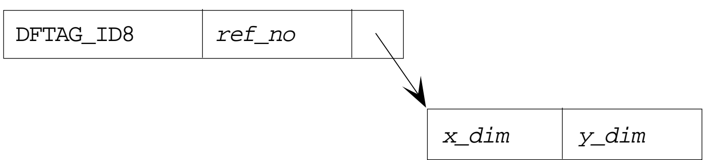
DFTAG_VS contains a block of data which is to be interpreted according to the information in the corresponding DFTAG_VH.

See also: DFTAG_VH (this section)
"Vsets, Vdatas, and Vgroups" in Chapter 4, "Sets and Groups"
NCSA HDF Vsets, Version 2.0 for HDF Version 3.2 and earlier
NCSA HDF User's Guide and *NCSA HDF Reference Manual* for HDF Version 3.3

Obsolete Tags

`DFTAG_ID8`

Image dimension-8
4 bytes
200 (0x00C8)



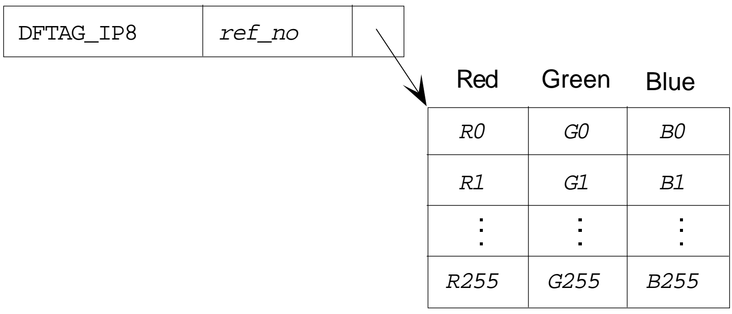
ref_no	Reference number (16-bit integer)
x_dim	Length of x dimension (16-bit integer)
y_dim	Length of y dimension (16-bit integer)

The data for this tag consists of two 16-bit integers representing the width and height of an 8-bit raster image in bytes.

This tag has been superseded by `DFTAG_ID`.

`DFTAG_IP8`

Image palette-8
768 bytes
201 (0x00C9)



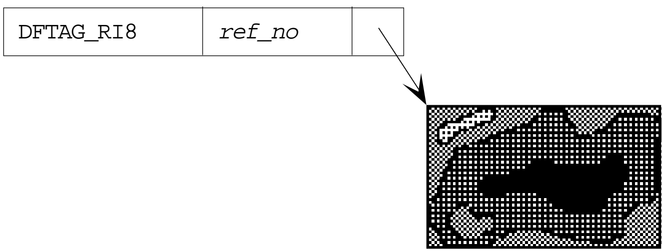
ref_no	Reference number (16-bit integer)
Table entries	256 triples of 8-bit integers

The data for this tag can be thought of as a table of 256 entries, each containing one value for red, green, and blue. The first triple is palette entry 0 and the last is palette entry 255.

This tag has been superseded by `DFTAG_LUT`.

DFTAG_RI8

Raster image-8
*xdim*ydim* bytes (where *xdim* and *ydim* are the dimensions specified in the corresponding DFTAG_ID8)
202 (0x00CA)



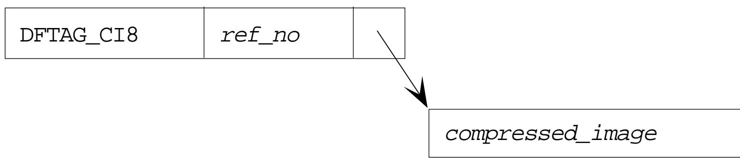
ref_no	Reference number (16-bit integer)
Image data	2-dimensional array of 8-bit integers

The data for this tag is a row-wise representation of the elementary 8-bit image data. The data is stored width-first (i.e., row-wise) and is 8 bits per pixel. The first byte of data represents the pixel in the upper-left hand corner of the image.

This tag has been superseded by DFTAG_RI.

DFTAG_CI8

Compressed image-8
? bytes
203 (0x00CB)



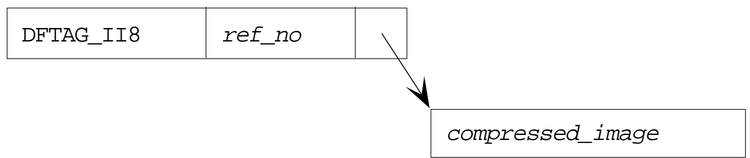
ref_no	Reference number (16-bit integer)
compressed_image	Series of run-length encoded bytes

The data for this tag is a row-wise representation of the elementary 8-bit image data. Each row is compressed using the following run-length encoding where *n* is the lower seven bits of the byte. The high bit indicates whether the following *n* bytes will be reproduced exactly (high bit = 0) or whether the following byte will be reproduced *n* times (high bit = 1). Since DFTAG_CI8 and DFTAG_RI8 are basically interchangeable, it is suggested that you not have a DFTAG_CI8 and a DFTAG_RI8 with the same reference number.

This tag has been superseded by DFTAG_RLE.

DFTAG_II8

IMCOMP image-8
? bytes
204 (0x00CC)



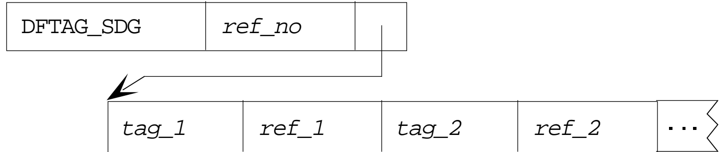
ref_no	Reference number (16-bit integer)
compressed_image	Compressed image data

The data for this tag is a 4:1 compressed 8-bit image, using the IMCOMP compression scheme.

This tag has been superseded by DFTAG_IMC.

DFTAG_SDG

Scientific data group
*n**4 bytes (where *n* is the number of data objects in the group)
700 (0x02BC)



ref_no Reference number (16-bit integer)

tag_n Tag number of nth member of the group (16-bit integer)

ref_n Reference number of nth member of the group (16-bit integer)

The SDG data element contains a list of tag/refs that define a scientific data set. All of the members of the group provide information required to correctly interpret and display the data. Application programs that deal with SDGs should read all of the elements of an SDG and process those which it can use. Even if an application cannot process all of the objects, the objects that it can understand will be usable.

Table 6.7 lists the tags that may appear in an SDG.

Table 6.7 Available SDG Tags

Tag	Description
DFTAG_SDD	Scientific data dimension record (rank and dimensions)
DFTAG_SD	Scientific data
DFTAG_SDS	Scales
DFTAG_SDL	Labels
DFTAG_SDU	Units
DFTAG_SDF	Formats
DFTAG_SDM	Maximum and minimum values
DFTAG_SDC	Coordinate system
DFTAG_SDT	Transposition (obsolete)
DFTAG_SDLNK	Link to new DFTAG_NDG

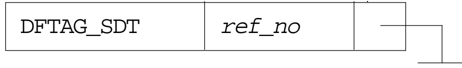
Example
DFTAG_SDD, DFTAG_SD, DFTAG_SDM
Assume that a dimension record, scientific data, and the maximum and minimum values of the data are required to read and interpret a particular data set. These data objects can be associated in an SDG so that an application can read the rank and dimensions from the dimension record and then read the data array. If the application needs the maximum and minimum values, it will read them as well.

This tag has been superseded by DFTAG_NDG**.**

See also: Chapter 4, “Sets and Groups”

DFTAG_SDT

Scientific data transpose
0 bytes
709 (0x02C5)



ref_no Reference number (16-bit integer)

The presence of this tag in a group indicates that the data pointed to by the corresponding DFTAG_SD is in column-major order, instead of the default row-major order. No data is associated with this tag.

This tag is no longer written by the HDF library. When it is encountered in an old file, it is interpreted as originally intended.

Chapter 7 Portability Issues

Chapter Overview

The NCSA implementation of HDF is accessible to both C and FORTRAN programs and is implemented on many different machines and several operating systems. There are important differences between C and FORTRAN, and among implementations of each language, especially FORTRAN. There are also important differences among the machines and operating systems that HDF supports.

If HDF is to be a portable tool, these differences must be constructively addressed. This chapter describes many of these differences, discusses the problems and issues associated with them, and presents the methods employed in the HDF implementation to reduce their impact.

The HDF Environment

The list of machines and operating systems on which HDF is implemented is steadily growing. For reasons that this chapter will make clear, the number of NCSA-supported HDF platforms is growing slowly. Every time a platform is added, additional code must be written to address concerns of memory management, operating system and file system differences, number representations, and differences in FORTRAN and C implementations on that system.

Supported Platforms

As of this writing, NCSA supports the platforms listed in Table 7.1.

Table 7.1 NCSA-supported HDF Platforms

Hardware Platform	Operating System
Convex	Concentrix
Cray X-MP, Y-MP, Cray 2	UNICOS
DEC Alpha	Ultrix
DECStation	Ultrix
HP 9000	HPUX
IBM PC	MS DOS, Windows 3.1
IBM RS/6000	AIX
IBM RT	UNIX
Macintosh	MPW Shell
NeXT	NeXTStep
Silicon Graphics	UNIX
Sun Sparc	UNIX
Vax	VMS

HDF has also been ported to several platforms that NCSA does not currently support. These include Alliant, Apollo (Domain), HP 3000, Stellar, Amiga, Symbolics, Fujitsu, and IBM 3090 (MVS).

Language Standards

Unfortunately, not all compilers are the same. FORTRAN compilers often differ in the ways they pass parameters, in the identifier naming conventions they employ, and in the number types that they support. Similarly, though generally not as drastically, C compilers differ in the number types that they support and in their adherence to the ANSI C standard.

To minimize the difficulties caused by these differences, the HDF source code is written primarily in the following dialects:

- FORTRAN 77
- ANSI C
- The original C defined by Kernighan and Ritchie[1], hereafter referred to as *old C*

Almost all platforms have C and FORTRAN compilers that adhere to at least one of these standards.

When time and resources permit, NCSA attempts to support features or variations in other dialects of C and FORTRAN, particularly on platforms that are important to NCSA users. Much of the remainder of this chapter addresses these efforts.

Guidelines

One cannot over stress the importance of following the guidelines outlined in this chapter. It may take longer to write code and it may be difficult to adapt your coding style, but the long-term benefits, in terms of portability and maintenance costs, will be well worth the effort.

---

1 The version of C described in the first edition of *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, published by Prentice-Hall.

Organization of Source Files

Three types of files appear in the HDF source code directory:

- Header files
- Source code files
- A makefile

Header files and source code files are organized by application area. All of the functions that apply to a particular application area are stored in three source files, and all the definitions and declarations that apply to that application are stored in a corresponding header file. The makefile describes the dependencies among the source and header files and provides the commands required to compile the corresponding libraries and utilities.

Header Files

Certain application modules require header files. The header file `dfan.h`, for example, contains definitions and declarations that are unique to the annotation interface.

There are also several general header files that are used in compiling the libraries for all application areas:

`hdf.h`, `hdfi.h`[2]

`hdf.h` contains declarations and definitions for the common data structures used throughout HDF, definitions of the HDF tags, definitions of error numbers, and definitions and declarations specific to the general purpose interface. Since `hdf.h` depends on `hdfi.h`, it includes `hdfi.h` via `#include`.

`hdfi.h` contains information specific to the various NCSA-supported HDF computing environments, environmental parameters that need to be set to particular values when compiling the HDF libraries, and machine dependent definitions of such things as number types and macros for reading and writing numbers.

When porting HDF to a new system, only `hdfi.h` and the makefile should need to be modified, though there may be exceptions.

It is normally a good idea to include `hdf.h` (and therefore indirectly `hdfi.h`) in user programs, though users usually need not be aware of its contents.

`hproto.h`

This file contains ANSI C prototypes for all HDF C routines. It must be included in ANSI C programs that call HDF routines.

`constants.i`

This file is for use in FORTRAN programs. It contains important constants, such as tag values, that are defined in hdf.h. Systems with FORTRAN preprocessors might be able to include this file via `#include` statements or their equivalent.

`dffunc.i`

This file is for use in FORTRAN programs. It contains declarations of all HDF FORTRAN-callable functions. Systems with FORTRAN preprocessors might be able to include this file via `#include` statements or their equivalent.

---

2 In earlier implementations of HDF, these files were called `df.h` and `dfi.h`. Starting with HDF Version 3.2, the general purpose layer of HDF was completely rewritten and all routine names were changed from `df*` to `hdf*`.

Source Code Files

All HDF operations are performed by routines written in C. Hence, even FORTRAN calls to HDF result in calls to the corresponding C routines. Because of the problems described below the relationships between the C routines and the corresponding FORTRAN routines can be confusing. This section discusses the C and FORTRAN source file organization. It is followed by discussions of problems users will face in the FORTRAN–C interface.

HDF interfaces typically have three or four associated files. For example, the scientific data set (SDS) interface is associated with the following files: `dfsd.h`, `dfsd.c`, `dfsdf.c`, and `dfsdff.f`.

These files fill the following roles:

Header files

The `*.h` files are header files.

Normal C routines

These routines do the actual HDF work. The others are used to transfer control and data from a FORTRAN environment to a C environment.

These routines are in the `*.c` files, as in `dfsd.c`. Every call to HDF, whether from C or FORTRAN, ultimately results in a call to one of these routines.

C routines that are directly callable from FORTRAN

These routines provide recognizable function names to the linker. They may also perform operations on data they receive from the FORTRAN routines that call them, such as transferring a FORTRAN string to a local C data area. Examples are provided below.

These routines are in the `*f.c` files, such as `dfsdf.c`. The `f` means that the routines can be called from FORTRAN; the `.c` means that they are C source code.

FORTRAN routines that perform some operation on the parameters that C would be unable to perform, before and/or after calling the corresponding C routine

These routines are required, for example, when one of the parameters is a string. The corresponding C routine has no way of knowing the length of the string unless it is explicitly given the length by the FORTRAN routine.

These routines are in the `*ff.f` files, such as `dfsdff.f`. The `ff` means that the routines perform some FORTRAN operation that C cannot perform and that they are to be called from FORTRAN; the `.f` means that they are FORTRAN source code.

The roles of these different types of source file types will become clearer as we look at some of the problems that arise in interfacing C and many different implementations of FORTRAN.

File naming conventions

The naming conventions for HDF library source code files are complicated by several factors. Because HDF must accommodate a wide variety of platforms, all files that will compile to object modules must have names that are unique in the first 8 characters, ignoring case. The difficulties involved in maintaining a FORTRAN-callable interface to a library that is primarily written in C further complicate the naming of source code files.

Passing Strings Between FORTRAN and C

One of the most important differences between FORTRAN and C compilers is in the way strings are represented. Different compilers use different data structures for strings, and supply string length information in different ways.

Passing Strings from FORTRAN to C

When strings are passed between FORTRAN and C routines, they may need to be converted from one representation to the other. C compilers store strings in an array of type `char`, terminated by a null byte (`\0`). The name of a string variable is equivalent to a pointer the first character in the string. FORTRAN compilers are not consistent in the ways that they store strings.

Two pieces of information must be acquired before FORTRAN can pass a string to C:
The string’s length
The string’s address

The string’s length is determined by invoking the standard FORTRAN function `len()`, which returns the length of a string. Since C expects a null byte at the end of a string, care must be taken that this null byte does not overwrite useful information in the FORTRAN string.

Determining the string’s address is more difficult because of the different ways that different FORTRAN implementations store strings. The macro `_fcdtocp` (FORTRAN character descriptor to C pointer) is used to acquire this information. `_fcdtocp` is one of the elements that must be customized for each platform. The following paragraphs discuss several existing customized implementations:

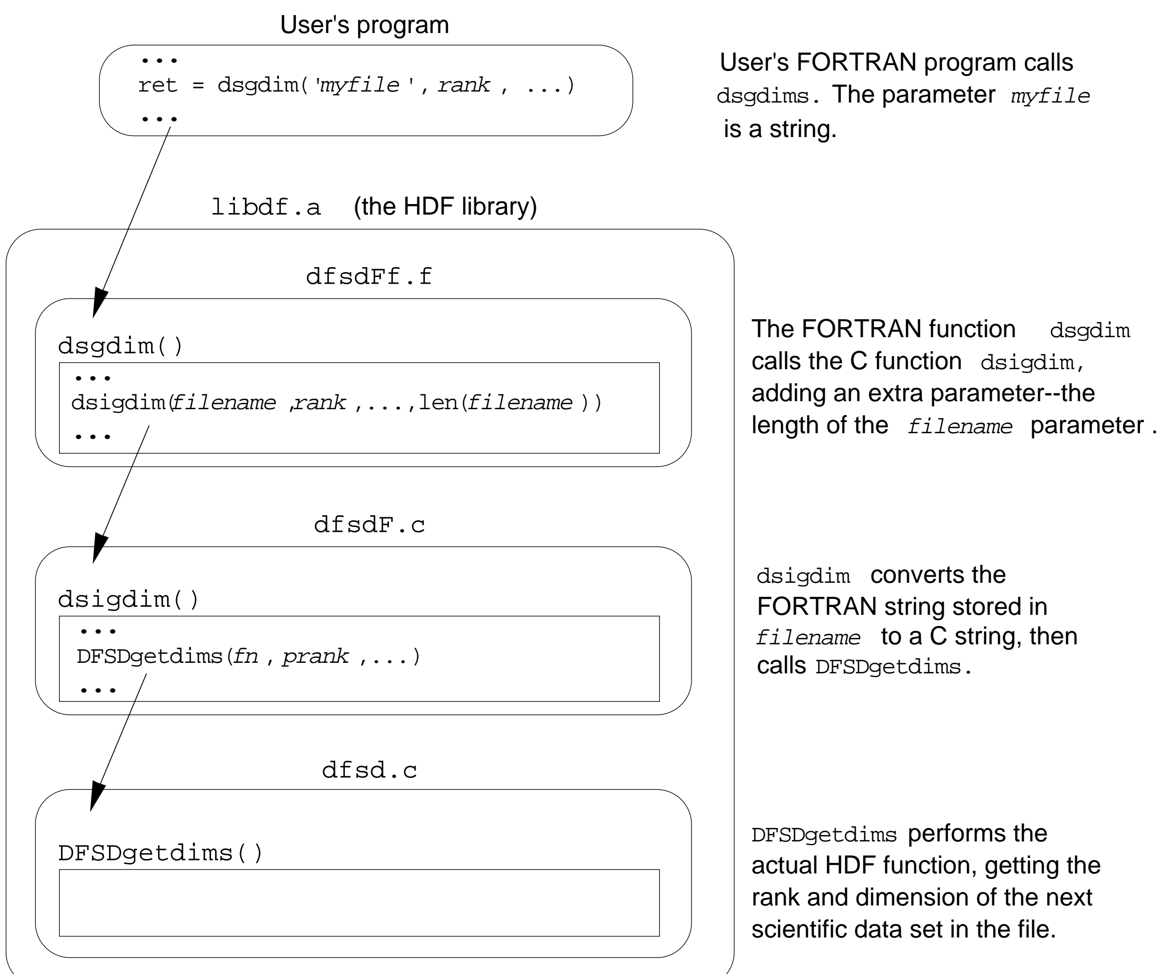
- UNICOS FORTRAN stores strings in a structure called `_fcd` (FORTRAN character descriptor). `_fcdtocp` is a built-in UNICOS function that returns the string’s address. (Since UNICOS provides this function, HDF omits the corresponding macro definition on UNICOS systems.)
- VMS FORTRAN uses a string descriptor structure that provides the string’s address and length. When compiled under VMS, `_fcdtocp` extracts the string's address from that structure.
- Most other FORTRAN compilers supported by HDF store strings just as C does, in character arrays with the array name identifying the array's address. In such situations, nothing special needs to be done to pass a string from FORTRAN to C, except to add a `NULL` byte..

An HDF FORTRAN call that involves passing a string results in the following sequences of actions:

1. A FORTRAN filter routine determines the length and address in memory of the string. Since this filter is a FORTRAN routine, it can be found in the appropriate `*ff.f` file.
2. The FORTRAN filter then calls a C routine, to which it passes all parameters from the initial call the string's length.
3. The C routine converts the FORTRAN string to a C string by copying it to a C array of type `char` and appending a null byte. Since this C routine serves as a link between a FORTRAN filter and the corresponding C interface call, it can be found in the appropriate `*f.c` file.
4. This C routine then calls the HDF C routine that performs the actual work.

This process is illustrated in Figure 7.1

Figure 7.1. Sequence of Events When a FORTRAN Call Includes a String as a Parameter



Passing Strings from C to FORTRAN

When strings are passed from C to FORTRAN, the reverse procedure is followed. First, a string pointer is allocated within the FORTRAN routine's data area. (It is assumed that the space pointed to has already been allocated, and is sufficiently large to hold the string.) The string is then copied from the C data area to the FORTRAN data area. Finally, the FORTRAN string's data area is padded with blanks, if necessary.

Function Return Values between FORTRAN and C

When a FORTRAN routine calls a C function, it always expects a return value from that function. Unfortunately, C functions do not always return arguments in a FORTRAN-compatible format.

To solve this problem, some FORTRAN compilers offer the option of controlling the form of the return value from a function. For example, Language Systems FORTRAN for the Macintosh requires that all C function declarations be prepended by the word `pascal` so that the return value can be recognized by a FORTRAN routine that calls it, as in:

```
pascal int dsgrang(void *pmax, void *pmin)
```

Since C always expects return values to be passed by value rather than, say, by reference, it is important to coerce FORTRAN functions to do the same. This is accomplished by defining a macro `FRETVAL` that is prepended to the declaration of every FORTRAN-callable C function. For example:

```
    FRETVAL(int)
dsgrang(void *pmax, void *pmin)
```

If Language Systems FORTRAN is to be used, `FRETVAL` is defined in `hdfi.h` as follows:

```
#if defined(MAC)        /* with LS FORTRAN */
#   define FRETVAL(x)   pascal x
#endif
```

Differences in Routine Names

HDF generally employs standard C conventions in naming routines. But many FORTRAN compilers impose varying restrictions on the length, character set, and form of identifiers, some of which are considerable more restrictive than the C conventions. Therefore, an extra effort must be made to accommodate those FORTRAN compilers.

To address this issue, HDF defines a set of preprocessor flags in `hdfi.h`. Then conditional compilation, with `#ifdef` statements in the source code , produces routine names that the target system’s FORTRAN will understand.

Case Sensitivity

C compilers are *case sensitive*; uppercase and lowercase letters are recognized as different characters. Many FORTRAN compilers are not case sensitive; they allow users to use uppercase and lowercase letters while naming routines in the source code, but the names are converted to all uppercase or all lowercase in the object module symbol tables. Routine name recognition problems are common when routines compiled by a case sensitive compiler are to be linked with routines compiled by a non-case sensitive compiler.

For example, the UNICOS FORTRAN compiler allows you to name routines without regard to case, but produces object module symbol tables with the routine names in all uppercase. UNICOS C, on the other hand, performs no such conversion.

Consider the HDF routine `Hopen`. `Hopen` is written in C, so the HDF library symbol table contains the name `Hopen`. Suppose you make the following call in your UNICOS FORTRAN program:

```
file_id = Hopen('myfile', ...)
```

The FORTRAN compiler will create an object module symbol table with the routine name `HOPEN`. When you link it to the HDF library, it will find `Hopen` but not `HOPEN`, and will generate an unsatisfied external reference error.

HDF supports the following non-case sensitive compilers:

- VMS FORTRAN
- UNICOS FORTRAN
- Language Systems FORTRAN.

All of these compilers convert identifiers to all uppercase when building an object module symbol table. In the following discussion, they are referred to as *all-uppercase compilers*.

The HDF Solution

HDF addresses the all-uppercase compiler problem in the platform-specific section of `hdfi.h` where the `DF_CAPFNAMES` flag is defined. With conditional compilation, HDF generates all-uppercase routine names and symbol table entries.

Once again, consider UNICOS. The UNICOS section of `hdfi.h` contains the following line:

```
#define DF_CAPFNAMES
```

The `*f.c` files contain corresponding conditional sections that produce all-uppercase routine names. For example, the function name `Fun` can be redefined as `FUN`:

```
#ifdef DF_CAPFNAMES
   define  Fun  FUN
#endif /* DF_CAPFNAMES */
```

Appended Underscores

Differing compiler conventions create a similar problem in their use of the underscore (_) character. Many compilers, including most C compilers, prepend an underscore to all external symbols in the object module symbol table. The linker then looks for external symbols in other symbol tables with the prefixed underscore.

Many FORTRAN compilers also *append* an underscore to identify external symbols. Since C compilers do not generally do this, external

references in FORTRAN-generated object modules will not recognize externals with the same names in C-generated modules.

For example, the FORTRAN compiler on the CONVEX system places an underscore both at the beginning and at the end of routine names, while the C compiler places an underscore only at the beginning.

Since `FUN` is a C function, it appears under the name `_FUN` in the object module containing it. Now suppose you make the following call in a FORTRAN program:

```
x = FUN(y)
```

The FORTRAN compiler will create an object module symbol table with the routine name `_FUN_`. When you link it to the C module, the linker will be unable to link `_FUN` and `_FUN_` and will generate an unsatisfied external reference error.

The HDF Solution

Like the all-uppercase compiler problem, this issue is resolved in the platform-specific sections of `hdfi.h` and with conditional sections of code that append an underscore to C routine names on platforms where the FORTRAN compiler expects it.

This is implemented as follows: The `FNAME_POST_UNDERSCORE` flag is defined in the platform-specific section of `hdfi.h` for every platform whose FORTRAN compiler requires appended underscores. Similarly, the `FNAME_PRE_UNDERSCORE` flag is defined on platforms where the FORTRAN compiler expects prepended underscores. The macro `FNAME` is then defined to append and/or prepend underscores as required.

The `FNAME` macro is then applied to each routine in the module in which it is actually defined (including in `hptroto.h`), adding the appropriate underscores.

Consider the above example in which `Fun` was renamed `FUN`. The actual definition appears as follows:

```
#ifdef DF_CAPFNAMES
   define  Fun  FNAME(FUN)
#endif /* DF_CAPFNAMES */
```

Short Names vs. Long Names

In the C implementations supported by HDF, identifiers may be any length with at least the first 31 characters being significant. FORTRAN compilers differ in the maximum lengths of identifiers that they allow, but all of those supported by HDF allow identifiers to be at least seven characters long.

To deal with the discrepancies between identifier lengths allowed by C and those allowed by the various FORTRAN compilers, a set of equivalent short names has been created for use when programming in FORTRAN. For every HDF routine with a name more than seven characters long, there is an identical routine whose name is seven or fewer characters long.

For example, the routines `DFSDgetdims` (in `dfsd.c`) and `dsgdims` (in `dfsdff.f`) are functionally identical.

Differences Between ANSI C and Old C

The current HDF release supports both ANSI C and old C compilers. ANSI C is preferred because it has many features that help ensure portability; unfortunately, many important platforms do not support full ANSI C. The HDF code determines whether ANSI C is available from the flag `__STDC__`. If ANSI C is available on a platform, then `__STDC__` is defined by the compiler.[3]

The most noticeable difference between ANSI C and old C is in the way functions are declared. For example, in ANSI C the function `DFSDsetdims()` is declared with a single line:

```
int DFSDsetdims(intn rank, int32 dimsizes[])
```

In old C the same function is declared as follows:

```
int DFSDsetdims(rank, dimsizes)
intn rank;
int32  dimsizes[];
```

HDF accommodates these differences by defining the flag `PROTOTYPE` in `hdfi.h`. `PROTOTYPE` is used for every function declaration in a manner similar to the following example:

```
#ifdef PROTOTYPE
int DFSDsetdims(intn rank, int32 dimsizes[])
#else
int DFSDsetdims(rank, dimsizes)
intn rank;
int32  dimsizes[];
#endif /* PROTOTYPE */
```

Note that prototypes are supported by some C compilers that are not otherwise ANSI-conformant. In such situations, `PROTOTYPE` is defined even though `__STDC__` is not.

Another difference between old C and ANSI C is that ANSI C supports function prototypes with arguments. (Old C also supports function prototypes, but without the argument list.) , This feature helps in detecting errors in the number and types of arguments. This difference is handled by means of a macro `PROTO`, which is defined as follows:

```
#ifdef PROTOTYPE
#define    PROTO(x) x
#else
#define    PROTO(x) ()
#endif
```

This macro is applied as in the following example:

```
extern int32 Hopen
PROTO((char *path, intn access, int16 ndds));
```

When `PROTOTYPE` is defined, `PROTO` causes the argument list to stay as it is. When `PROTOTYPE` is not defined, `PROTO` causes the argument list to disappear.

3 `__STDC__` is generally defined by ANSI-conforming C compilers. Some C compilers are not entirely ANSI-conforming, yet they conform well enough that the HDF implementation can treat them as if they were. In such cases, it is permissible to define `__STDC__` by adding the option `-D__STDC__` to the `cc` line in the makefile.

Type Differences

Platforms and compilers also differ in the sizes of numbers that they assign to different data types, in their representations of different number types, and in the way they organize aggregates of numbers (especially structures).

Size differences

The same number type can be different sizes on different platforms. The type `int`, for example, is 16 bits to many IBM PC compilers, 48 bits to some supercomputer compilers, and 32 bits on most others. This can cause problems that are difficult to diagnose in code, like the HDF code, that depends in many places on numbers being the right size.

HDF handles this problem by fully defining all variable types and function data types via `typedef`, including the number of bits occupied. All parameters, members of structures, and static, automatic, and external variables are so defined .

The HDF data types include the following (types with the prefix `u` are unsigned.)

int8
uint8
int16
uint16
int32
uint32
float32
float64
intn
uintn

For each machine, typedefs are declared that map all of the data types used into the best available types. For example, `int32` is defined as follows for Sun's C compiler:

```
typedef long int int32;
```

Unfortunately, the HDF data types do not always map exactly to one of the native data types. For example, the Cray UNICOS C compiler does not support a 16-bit data type. In such instances, HDF uses the best available match and care is taken to minimize potential problems.

The data types `intn` and `uintn` are for situations where it can be determined that number type size is unimportant and that a 16-bit integer is large enough to hold any value the number can have. In such cases, the native integer type (or unsigned integer type) of the host machine is used. Experience indicates that substantial performance gains can be achieved by using `intn` or `uintn` in certain circumstances.

Number Representation

One of the keys to producing a portable file format is to ensure that numbers that are represented differently on different machines are converted correctly when moved from machine to machine. HDF provides conversion routines to convert between native representations and a standard representation that is actually used in the HDF file. This ensures that HDF data will always be interpreted correctly, regardless of the platform on which it is read or written. Details of this process will be included in a later edition of this manual.

Byte-order and Structure Representations

Even when the basic bit-representation of constants or aggregates like structures is the same across platforms, the ways that the bits are packed into a word and the order in which the bits are laid out can differ. For example, DEC and Intel-based machines generally order bytes differently from most others. And the C compiler on a Cray, with a 64-bit word, packs structures differently from those on 32-bit word machines.

Differences in byte order among machines are handled in either of two ways. When the data to be written (or read) includes non-integer data and/or a large array of any type of data, conversion routines mentioned in the previous section, "Number Representation," are invoked. When an individual integer is to be written (or read), an `ENCODE` or `DECODE` macro is used.

The following ENCODE and DECODE macros are available for 16-bit and 32-bit integers:

```
INT16ENCODE
UINT16ENCODE
INT32ENCODE
UINT32ENCODE
INT16DECODE
UINT16DECODE
INT32DECODE
UINT32DECODE
```

The ENCODE macros write integers to an HDF file in a standard format regardless of the word-size and byte order of the host machine.

Likewise, the DECODE macros read integers from a standard format in an HDF file and provide the integers in the required byte order and word size to the host machine.

Since the ENCODE and DECODE macros deal with both byte order and word size, they are also used in reading and writing record-like structures. For example, an HDF data descriptor consists of two 16-bit fields followed by two 32-bit fields, as implied by the following C declaration:

```
struct {
  uint16 tag;
  uint16 ref;
  uint32 offset;
  uint32 length;
}
```

Even though this structure might occupy 12 bytes on one platform or 32 bytes on another (e.g., a Cray), it must occupy exactly 12 bytes in an HDF file. Furthermore, some machines represent the numbers internally in different byte orders than others, but the byte order must always be big-endian in an HDF file. The ENCODE and DECODE

macros ensure that these values are always represented correctly in HDF files and as presented to any host machine.

Access to Library Functions

Despite standardization efforts, function libraries often differ in significant ways. At least three types of functions require special treatment in the HDF implementation:

File I/O
: Some platforms use 16-bit values for the element size and the number of elements to write or read, while others use 32-bit values. This must be considered when working with either stream or system level I/O functions (i.e., the functions associated with the `fopen()` and `open()` calls).

Memory allocation and release
: First, 16-bit machines use a 16-bit value to indicate the number of bytes to allocate or release at one time. Second, certain operating systems (notably MS Windows and MAC/OS) don't have `malloc()` and `free()` calls. These operating systems use handles for allocating memory and require different function calls.

Memory and string manipulation
: These functions (e.g., `memcpy()`, `memcmp()`, `strcpy()`, and `strlen()`) require slightly different function names under different memory models in MS DOS and under MS Windows than on most other systems.

HDF accommodates these special situations by defining appropriate macros in the machine-specific sections of `hdfi.h`.

Appendix A Tags and Extended Tag Labels

The tables in this appendix lists all of the NCSA-supported HDF tags and the labels used to identify extended tags.

Tags

Table A.1 lists all the NCSA-supported HDF tags with the following information:

Tag — The tag itself

Tag number — The regular tag number in decimal (top) and hexadecimal (bottom)

Extended tag number — The extended tag number used with linked blocks and external data elements in decimal and (hexadecimal)

Full name — The tag name, a descriptive English phrase

Section — The section of Chapter 6, “Tag Specifications,” in which the tag is discussed

Table A.1 NCSA-supported HDF Tags

Tag	Number	Extended Number	Full Name	Section
DFTAG_AR	312 0x0138		Aspect ratio	Raster Image Tags
DFTAG_CAL	731 0x02DB		Calibration information	Scientific Data Set Tags
DFTAG_CCN	310 0x0136		Color correction	Raster Image Tags
DFTAG_CFM	311 0x0137		Color format	Raster Image Tags
DFTAG_CI8	203 0x00CB		Compressed image-8	Obsolete Tags
DFTAG_DIA	105 0x0069		Data identifier annotation	Annotation Tags

Table A.1 NCSA-supported HDF Tags *(Continued)*

Tag	Number	Extended Number	Full Name	Section
DFTAG_DIL	104 0x0068		Data identifier label	Annotation Tags
DFTAG_DRAW	400 0x0190		Draw	Composite Image Tags
DFTAG_FD	101 0x0065		File description	Annotation Tags
DFTAG_FID	100 0x0064		File identifier	Annotation Tags
DFTAG_FV	732 0x02DC		Fill value	Scientific Data Set Tags
DFTAG_GREYJPEG	14 0x000E		8-bit JPEG compression information	Compression Tags
DFTAG_ID	300 0x012C		Image dimension	Raster Image Tags
DFTAG_ID8	200 0x00C8		Image dimension-8	Obsolete Tags
DFTAG_II8	204 0x00CC		IMCOMP image-8	Obsolete Tags
DFTAG_IMC	12 0x000C		IMCOMP compressed data	Compression Tags
DFTAG_IP8	201 0x00C9		Image palette-8	Obsolete Tags
DFTAG_JPEG	13 0x000D		24-bit JPEG compression information	Compression Tags
DFTAG_LD	307 0x0133		LUT dimension	Raster Image Tags
DFTAG_LUT	301 0x012D		Lookup table	Raster Image Tags
DFTAG_MA	309 0x0135		Matte channel	Raster Image Tags
DFTAG_MD	308 0x0134		Matte channel dimension	Raster Image Tags
DFTAG_MT	107 0x006B		Machine type	Utility Tags
DFTAG_NDG	720 0x02D0		Numeric data group	Scientific Data Set Tags
DFTAG_NT	106 0x006A		Number type	Utility Tags
DFTAG_NULL	1 0x0001		No data	Utility Tags
DFTAG_RI	302 0x012E	16686 0x412E	Raster image	Raster Image Tags
DFTAG_RI8	202 0x00CA		Raster image-8	Obsolete Tags

Table A.1 NCSA-supported HDF Tags *(Continued)*

Tag	Number	Extended Number	Full Name	Section
DFTAG_RIG	306 0x0132		Raster image group	Raster Image Tags
DFTAG_RLE	11 0x000B		Run length encoded data	Compression Tags
DFTAG_SD	702 0x02BE	17086 0x42BE	Scientific data	Scientific Data Set Tags
DFTAG_SDC	708 0x02C4		Scientific data coordinates	Scientific Data Set Tags
DFTAG_SDD	701 0x02BD		Scientific data dimension record	Scientific Data Set Tags
DFTAG_SDF	706 0x02C2		Scientific data format	Scientific Data Set Tags
DFTAG_SDG	700 0x02BC		Scientific data group	Obsolete Tags
DFTAG_SDL	704 0x02C0		Scientific data labels	Scientific Data Set Tags
DFTAG_SDLNK	710 0x02C6		Scientific data set link	Scientific Data Set Tags
DFTAG_SDM	707 0x02C3		Scientific data max/min	Scientific Data Set Tags
DFTAG_SDS	703 0x02BF		Scientific data scales	Scientific Data Set Tags
DFTAG_SDT	709 0x02C5		Scientific data transpose	Obsolete Tags
DFTAG_SDU	705 0x02C1		Scientific data units	Scientific Data Set Tags
DFTAG_T105	603 0x25B		Tektronix 4105	Vector Image Tags
DFTAG_T14	602 0x25A		Tektronix 4014	Vector Image Tags
DFTAG_TD	103 0x0067		Tag description	Annotation Tags
DFTAG_TID	102 0x0066		Tag identifier	Annotation Tags
DFTAG_VERSION	30 0x001E		Library version number	Utility Tags
DFTAG_VG	1965 0x07AD		Vgroup	Vset Tags
DFTAG_VH	1962 0x07AA		Vdata description	Vset Tags
DFTAG_VS	1963 0x07AB	18347 0x47AB	Vdata	Vset Tags
DFTAG_XYP	500 0x01F4		X-Y position	Composite Image Tags

Extended Tag Labels

Table A.2 lists labels used to identify HDF extended tags. The table includes the following information:

Extended tag label
: The label, which appears as the first element of the extended tag description record

Physical storage method
: The alternative storage method indicated by the label

Table A.2 Extended Tag Labels

Extended Tag Label	Physical Storage Method
`EXT_EXTERN`	External file element
`EXT_LINKED`	Linked block element

Preferences:

? All numbers shown in DECIMAL (if not stated otherwise)
? First byte in a file has No: 0000
? Abbreviations : hex. = hexadecimal, dec. = decimal
? LSB = least significant byte
? MSB = most significant Byte
? NN = value varies

Structure of a PCS file:

***** Header *****

Byte (hex)	Value (hex)	Meaning
0000	32	lead-in char., always hex 32
0001	02 or 03	small hoop = 02 large = 03
0002-0003	10 00	No. of colors ALWAYS 16
0004-0043	NN NN NN NN	color definition, 4-Byte per color. 16 colors
0044-0045	NN NN	Nr. of stitches in file, LSB first. Max. 65536 Stitch count does NOT (!) include color changes

***** Stitches and color change records *****
***** each nine bytes long *****

0046-NNNN

***** stitch record *****

NNNN-NNNN 00 XX XX XX 00 YY YY YY 00 x and y
-or- coordinates
NNNN-NNNN 00 XX XX XX 00 YY YY YY 02 LSB first

***** color change record *****

NNNN-NNNN 02 00 00 00 00 00 00 00 03

First Byte gives the appropriate number of the color, Last Byte = 03 marks record as color change,

***** Bitmap file-name and design description *****

Def.: MMMM = (stitch count + No. of color changes) * 9

0046+MMMM Bitmap filename used as pattern for digitizing (12 characters) 00 terminated

Def.: DDDD = MMMM + 13

0046+DDDD File description, 00 terminated

This concludes the structure description of a typical PCS file. The Hex dump of a 2-stitch, 1-color change PCS file designed for the 80x80 hoop (small) with the bitmap pattern ëFEDER.BMPí and the file description ëSchnatzka Bubublickí follows:

Hex-dump of a 2-stitch design file:

Byte 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

```
0000  32 02 10 00  00 00 00 00  00 00 80 00  00 00 FF 00
0010  00 80 80 00  00 FF FF 00  80 00 80 00  FF 00 FF 00
0020  80 00 00 00  FF 00 00 00  00 80 00 00  00 FF 00 00
0030  80 80 00 00  FF FF 00 00  80 80 80 00  C0 C0 C0 00
0040  FF FF FF 00  02 00 00 00  00 00 00 00  00 00 03 00
0050  73 00 00 00  27 01 00 00  00 33 01 00  00 81 00 00
0060  00 46 45 44  45 52 2E 42  4D 50 00 00  00 00 53 63
0070  68 6E 61 74  7A 6B 61 20  42 75 62 75  62 6C 69 63
0080  6B 00
```

Here¥s the formatted TEXT dump of the same file:

```
Byte  0123456789ABCDEF

0000  2  ∑∑∑∑∑∑∑«∑∑∑_∑
0010  ∑««∑∑__∑«∑«∑_∑_∑
0020  «∑∑∑_∑∑∑∑«∑∑∑_∑∑
0030  ««∑∑__∑∑«««∑+++∑
0040  ___∑ ∑∑∑∑∑∑∑∑∑ ∑
0050  s∑∑∑' ∑∑∑3 ∑∑¸∑∑
0060  ∑FEDER.BMP∑∑∑∑Sc
0070  hnatzka Bubublic
0080  k∑
```

INTERNET-DRAFT T. Boutell, et al
Portable Network Graphics Boutell.Com, Inc.
Expires: 10 Dec 1996 10 June 1996

PNG (Portable Network Graphics) Specification

Version 1.0

File draft-boutell-png-spec-04.txt

Status of This Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

To learn the current status of any Internet-Draft, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or ftp.isi.edu (US West Coast).

Distribution of this memo is unlimited.

Abstract

This document describes PNG (Portable Network Graphics), an extensible file format for the lossless, portable, well-compressed storage of raster images. PNG provides a patent-free replacement for GIF and can also replace many common uses of TIFF. Indexed-color, grayscale, and truecolor images are supported, plus an optional alpha channel. Sample depths range from 1 to 16 bits.

PNG is designed to work well in online viewing applications, such as the World Wide Web, so it is fully streamable with a progressive display option. PNG is robust, providing both full file integrity checking and simple detection of common transmission errors. Also, PNG can store gamma and chromaticity data for improved color matching on heterogeneous platforms.

This specification defines a proposed Internet Media Type image/png.

Table of Contents

1. Introduction 4
2. Data Representation 5
   2.1. Integers and byte order 5
   2.2. Color values 5
   2.3. Image layout 6
   2.4. Alpha channel 7

2.5. Filtering 8
2.6. Interlaced data order 8
2.7. Gamma correction 9
2.8. Text strings 10
3. File Structure 10
3.1. PNG file signature 10
3.2. Chunk layout 10
3.3. Chunk naming conventions 11
3.4. CRC algorithm 14
4. Chunk Specifications 14
4.1. Critical Chunks 14
4.1.1. IHDR Image Header 14
4.1.2. PLTE Palette 16
4.1.3. IDAT Image Data 16
4.1.4. IEND Image Trailer 17
4.2. Ancillary Chunks 17
4.2.1. bKGD Background Color 17
4.2.2. cHRM Primary Chromaticities and White Point 18
4.2.3. gAMA Image Gamma 19
4.2.4. hIST Image Histogram 19
4.2.5. pHYs Physical Pixel Dimensions 20
4.2.6. sBIT Significant Bits 21
4.2.7. tEXt Textual Data 22
4.2.8. tIME Image Last-Modification Time 23
4.2.9. tRNS Transparency 24
4.2.10. zTXt Compressed Textual Data 25
4.3. Summary of Standard Chunks 26
4.4. Additional Chunk Types 26
5. Deflate/Inflate Compression 27
6. Filter Algorithms 29
6.1. Filter type 0: None 30
6.2. Filter type 1: Sub 30
6.3. Filter type 2: Up 31
6.4. Filter type 3: Average 31
6.5. Filter type 4: Paeth 32
7. Chunk Ordering Rules 33
7.1. Behavior of PNG editors 34
7.2. Ordering of ancillary chunks 34
7.3. Ordering of critical chunks 35
8. Miscellaneous Topics 35
8.1. File name extension 35
8.2. Internet media type 35

8.3. Macintosh file layout 35
8.4. Multiple-image extension 36
8.5. Security considerations 36
9. Recommendations for Encoders 37
9.1. Bit depth scaling 37
9.2. Encoder gamma handling 39
9.3. Encoder color handling 41
9.4. Alpha channel creation 43
9.5. Suggested palettes 43
9.6. Filter selection 44
9.7. Text chunk processing 44
9.8. Use of private chunks 45
9.9. Private type and method codes 46
10. Recommendations for Decoders 46
10.1. Error checking 46
10.2. Pixel dimensions 47
10.3. Truecolor image handling 47
10.4. Bit depth rescaling 48
10.5. Decoder gamma handling 49
10.6. Decoder color handling 51
10.7. Background color 52
10.8. Alpha channel processing 52
10.9. Progressive display 56

10.10. Suggested-palette and histogram usage 57
10.11. Text chunk processing 58
11. Glossary 59
12. Appendix: Rationale 63
12.1. Why a new file format? 63
12.2. Why these features? 63
12.3. Why not these features? 64
12.4. Why not use format X? 65
12.5. Byte order 65
12.6. Interlacing 66
12.7. Why gamma? 66
12.8. Non-premultiplied alpha 67
12.9. Filtering 68
12.10. Text strings 68
12.11. PNG file signature 69
12.12. Chunk layout 70
12.13. Chunk naming conventions 70
12.14. Palette histograms 72
13. Appendix: Gamma Tutorial 72
14. Appendix: Color Tutorial 80
15. Appendix: Sample CRC Code 84
16. Appendix: Online Resources 86
17. Appendix: Revision History 87
18. References 87
19. Credits 89

1. Introduction

The PNG format provides a portable, legally unencumbered, well-compressed, well-specified standard for lossless bitmapped image files.

Although the initial motivation for developing PNG was to replace GIF, the design provides some useful new features not available in GIF, with minimal cost to developers.

GIF features retained in PNG include:

* Indexed-color images of up to 256 colors.
* Streamability: files can be read and written serially, thus allowing the file format to be used as a communications protocol for on-the-fly generation and display of images.
* Progressive display: a suitably prepared image file can be displayed as it is received over a communications link, yielding a low-resolution image very quickly followed by gradual improvement of detail.
* Transparency: portions of the image can be marked as transparent, creating the effect of a nonrectangular image.
* Ancillary information: textual comments and other data can be stored within the image file.
* Complete hardware and platform independence.
* Effective, 100% lossless compression.

Important new features of PNG, not available in GIF, include:

* Truecolor images of up to 48 bits per pixel.
* Grayscale images of up to 16 bits per pixel.
* Full alpha channel (general transparency masks).
* Image gamma information, which supports automatic display of images with correct brightness/contrast regardless of the machines used to originate and display the image.
* Reliable, straightforward detection of file corruption.

* Faster initial presentation in progressive display mode.

PNG is designed to be:

* Simple and portable: developers should be able to implement PNG easily.
* Legally unencumbered: to the best knowledge of the PNG authors, no algorithms under legal challenge are used. (Some considerable effort has been spent to verify this.)
* Well compressed: both indexed-color and truecolor images are compressed as effectively as in any other widely used lossless format, and in most cases more effectively.
* Interchangeable: any standard-conforming PNG decoder will read all conforming PNG files.
* Flexible: the format allows for future extensions and private add-ons, without compromising interchangeability of basic PNG.

* Robust: the design supports full file integrity checking as well as simple, quick detection of common transmission errors.

The main part of this specification gives the definition of the file format and recommendations for encoder and decoder behavior. An appendix gives the rationale for many design decisions. Although the rationale is not part of the formal specification, reading it can help implementors understand the design. Cross-references in the main text point to relevant parts of the rationale. Additional appendixes, also not part of the formal specification, provide tutorials on gamma and color theory as well as other supporting material.

See Rationale: Why a new file format? (Section 12.1), Why these features? (Section 12.2), Why not these features? (Section 12.3), Why not use format X? (Section 12.4).

Pronunciation

PNG is pronounced "ping".

2. Data Representation

This chapter discusses basic data representations used in PNG files, as well as the expected representation of the image data.

2.1. Integers and byte order

All integers that require more than one byte will be in network byte order: the most significant byte comes first, then the less significant bytes in descending order of significance (MSB LSB for two-byte integers, B3 B2 B1 B0 for four-byte integers). The highest bit (value 128) of a byte is numbered bit 7; the lowest bit (value 1) is numbered bit 0. Values are unsigned unless otherwise noted. Values explicitly noted as signed are represented in two's complement notation.

See Rationale: Byte order (Section 12.5).

2.2. Color values

Colors can be represented by either grayscale or RGB (red, green, blue) sample data. Grayscale data represents luminance; RGB data represents calibrated color information (if the cHRM chunk is present) or uncalibrated device-dependent color (if cHRM is absent). All color values range from zero (representing black) to most intense at the maximum value for the bit depth. Note that the maximum value at a given bit depth is (2^bitdepth)-1, not 2^bitdepth.

Sample values are not necessarily linear; the gAMA chunk specifies the gamma characteristic of the source device, and viewers are

strongly encouraged to compensate properly. See Gamma correction (Section 2.7).

Source data with a precision not directly supported in PNG (for example, 5 bit/sample truecolor) must be scaled up to the next higher supported bit depth. This scaling is reversible with no loss of data, and it reduces the number of cases that decoders must cope with. See Recommendations for Encoders: Bit depth scaling (Section 9.1) and Recommendations for Decoders: Bit depth rescaling (Section 10.4).

2.3. Image layout

Conceptually, a PNG image is a rectangular pixel array, with pixels appearing left-to-right within each scanline, and scanlines appearing top-to-bottom. (For progressive display purposes, the data may actually be transmitted in a different order; see Interlaced data order, Section 2.6.) The size of each pixel is determined by the bit depth, which is the number of bits per sample in the image data.

Three types of pixel are supported:

* An indexed-color pixel is represented by a single sample that is an index into a supplied palette. The image bit depth determines the maximum number of palette entries, but not the color precision within the palette.
* A grayscale pixel is represented by a single sample that is a grayscale level, where zero is black and the largest value for the bit depth is white.
* A truecolor pixel is represented by three samples: red (zero = black, max = red) appears first, then green (zero = black, max = green), then blue (zero = black, max = blue). The bit depth specifies the size of each sample, not the total pixel size.

Optionally, grayscale and truecolor pixels can also include an alpha sample, as described in the next section.

Pixels are always packed into scanlines with no wasted bits between pixels. Pixels smaller than a byte never cross byte boundaries; they are packed into bytes with the leftmost pixel in the high-order bits of a byte, the rightmost in the low-order bits. Permitted bit depths and pixel types are restricted so that in all cases the packing is simple and efficient.

PNG permits multi-sample pixels only with 8- and 16-bit samples, so multiple samples of a single pixel are never packed into one byte. 16-bit samples are stored in network byte order (MSB first).

Scanlines always begin on byte boundaries. When pixels have fewer

than 8 bits and the scanline width is not evenly divisible by the number of pixels per byte, the low-order bits in the last byte of

each scanline are wasted. The contents of these wasted bits are unspecified.

An additional "filter type" byte is added to the beginning of every scanline (see Filtering, Section 2.5). The filter type byte is not considered part of the image data, but it is included in the datastream sent to the compression step.

2.4. Alpha channel

An alpha channel, representing transparency information on a per-pixel basis, can be included in grayscale and truecolor PNG images.

An alpha value of zero represents full transparency, and a value of (2^bitdepth)-1 represents a fully opaque pixel. Intermediate values indicate partially transparent pixels that can be combined with a background image to yield a composite image. (Thus, alpha is really the degree of opacity of the pixel. But most people refer to alpha as providing transparency information, not opacity information, and we continue that custom here.)

Alpha channels can be included with images that have either 8 or 16 bits per sample, but not with images that have fewer than 8 bits per sample. Alpha samples are represented with the same bit depth used for the image samples. The alpha sample for each pixel is stored immediately following the grayscale or RGB samples of the pixel.

The color values stored for a pixel are not affected by the alpha value assigned to the pixel. This rule is sometimes called "unassociated" or "non-premultiplied" alpha. (Another common technique is to store sample values premultiplied by the alpha fraction; in effect, such an image is already composited against a black background. PNG does not use premultiplied alpha.)

Transparency control is also possible without the storage cost of a full alpha channel. In an indexed-color image, an alpha value can be defined for each palette entry. In grayscale and truecolor images, a single pixel value can be identified as being "transparent". These techniques are controlled by the tRNS ancillary chunk type.

If no alpha channel nor tRNS chunk is present, all pixels in the image are to be treated as fully opaque.

Viewers can support transparency control partially, or not at all.

See Rationale: Non-premultiplied alpha (Section 12.8), Recommendations for Encoders: Alpha channel creation (Section

9.4), and Recommendations for Decoders: Alpha channel processing (Section 10.8).

2.5. Filtering

PNG allows the image data to be filtered before it is compressed. Filtering can improve the compressibility of the data. The filter step itself does not reduce the size of the data. All PNG filters are strictly lossless.

PNG defines several different filter algorithms, including "none" which indicates no filtering. The filter algorithm is specified for each scanline by a filter type byte that precedes the filtered scanline in the precompression datastream. An intelligent encoder can switch filters from one scanline to the next. The method for

choosing which filter to employ is up to the encoder.

See Filter Algorithms (Chapter 6) and Rationale: Filtering (Section 12.9).

2.6. Interlaced data order

A PNG image can be stored in interlaced order to allow progressive display. The purpose of this feature is to allow images to "fade in" when they are being displayed on-the-fly. Interlacing slightly expands the file size on average, but it gives the user a meaningful display much more rapidly. Note that decoders are required to be able to read interlaced images, whether or not they actually perform progressive display.

With interlace method 0, pixels are stored sequentially from left to right, and scanlines sequentially from top to bottom (no interlacing).

Interlace method 1, known as Adam7 after its author, Adam M. Costello, consists of seven distinct passes over the image. Each pass transmits a subset of the pixels in the image. The pass in which each pixel is transmitted is defined by replicating the following 8-by-8 pattern over the entire image, starting at the upper left corner:

```
1 6 4 6 2 6 4 6
7 7 7 7 7 7 7 7
5 6 5 6 5 6 5 6
7 7 7 7 7 7 7 7
3 6 4 6 3 6 4 6
7 7 7 7 7 7 7 7
5 6 5 6 5 6 5 6
7 7 7 7 7 7 7 7
```

Within each pass, the selected pixels are transmitted left to right within a scanline, and selected scanlines sequentially from

top to bottom. For example, pass 2 contains pixels 4, 12, 20, etc. of scanlines 0, 8, 16, etc. (numbering from 0,0 at the upper left corner). The last pass contains the entirety of scanlines 1, 3, 5, etc.

The data within each pass is laid out as though it were a complete image of the appropriate dimensions. For example, if the complete image is 16 by 16 pixels, then pass 3 will contain two scanlines, each containing four pixels. When pixels have fewer than 8 bits, each such scanline is padded as needed to fill an integral number of bytes (see Image layout, Section 2.3). Filtering is done on this reduced image in the usual way, and a filter type byte is transmitted before each of its scanlines (see Filter Algorithms, Chapter 6). Notice that the transmission order is defined so that all the scanlines transmitted in a pass will have the same number of pixels; this is necessary for proper application of some of the filters.

Caution: If the image contains fewer than five columns or fewer than five rows, some passes will be entirely empty. Encoder and decoder authors must be careful to handle this case correctly. In particular, filter type bytes are only associated with nonempty scanlines; no filter type bytes are present in an empty pass.

See Rationale: Interlacing (Section 12.6) and Recommendations for Decoders: Progressive display (Section 10.9).

2.7. Gamma correction

PNG images can specify, via the gAMA chunk, the gamma characteristic of the image with respect to the original scene. Display programs are strongly encouraged to use this information, plus information about the display device they are using and room lighting, to present the image to the viewer in a way that reproduces what the image's original author saw as closely as possible. See Gamma Tutorial (Chapter 13) if you aren't already familiar with gamma issues.

Gamma correction is not applied to the alpha channel, if any. Alpha samples always represent a linear fraction of full opacity.

For high-precision applications, the exact chromaticity of the RGB data in a PNG image can be specified via the cHRM chunk, allowing more accurate color matching than gamma correction alone will provide. See Color Tutorial (Chapter 14) if you aren't already familiar with color representation issues.

See Rationale: Why gamma? (Section 12.7), Recommendations for Encoders: Encoder gamma handling (Section 9.2), and Recommendations for Decoders: Decoder gamma handling (Section 10.5).

2.8. Text strings

A PNG file can store text associated with the image, such as an image description or copyright notice. Keywords are used to indicate what each text string represents.

ISO 8859-1 (Latin-1) is the character set recommended for use in text strings [ISO-8859]. This character set is a superset of 7-bit ASCII.

Character codes not defined in Latin-1 should not be used, because they have no platform-independent meaning. If a non-Latin-1 code does appear in a PNG text string, its interpretation will vary across platforms and decoders. Some systems might not even be able to display all the characters in Latin-1, but most modern systems can.

Provision is also made for the storage of compressed text.

See Rationale: Text strings (Section 12.10).

3. File Structure

A PNG file consists of a PNG signature followed by a series of chunks. This chapter defines the signature and the basic properties of chunks. Individual chunk types are discussed in the next chapter.

3.1. PNG file signature

The first eight bytes of a PNG file always contain the following (decimal) values:

```
137 80 78 71 13 10 26 10
```

This signature indicates that the remainder of the file contains a single PNG image, consisting of a series of chunks beginning with an IHDR chunk and ending with an IEND chunk.

See Rationale: PNG file signature (Section 12.11).

3.2. Chunk layout

Each chunk consists of four parts:

Length
A 4-byte unsigned integer giving the number of bytes in the chunk's data field. The length counts only the data field, not itself, the chunk type code, or the CRC. Zero is a valid length. Although encoders and decoders should treat the length as unsigned, its value must not exceed (2^31)-1 bytes.

Chunk Type
A 4-byte chunk type code. For convenience in description and in examining PNG files, type codes are restricted to consist of uppercase and lowercase ASCII letters (A-Z and a-z, or 65-90 and 97-122 decimal). However, encoders and decoders must treat the codes as fixed binary values, not character strings. For example, it would not be correct to represent the type code IDAT by the EBCDIC equivalents of those letters. Additional naming conventions for chunk types are discussed in the next section.

Chunk Data
The data bytes appropriate to the chunk type, if any. This field can be of zero length.

CRC
A 4-byte CRC (Cyclic Redundancy Check) calculated on the preceding bytes in the chunk, including the chunk type code and chunk data fields, but not including the length field. The CRC is always present, even for chunks containing no data. See CRC algorithm (Section 3.4).

The chunk data length can be any number of bytes up to the maximum; therefore, implementors cannot assume that chunks are aligned on any boundaries larger than bytes.

Chunks can appear in any order, subject to the restrictions placed on each chunk type. (One notable restriction is that IHDR must appear first and IEND must appear last; thus the IEND chunk serves as an end-of-file marker.) Multiple chunks of the same type can appear, but only if specifically permitted for that type.

See Rationale: Chunk layout (Section 12.12).

3.3. Chunk naming conventions

Chunk type codes are assigned so that a decoder can determine some properties of a chunk even when it does not recognize the type code. These rules are intended to allow safe, flexible extension of the PNG format, by allowing a decoder to decide what to do when it encounters an unknown chunk. The naming rules are not normally of interest when the decoder does recognize the chunk's type.

Four bits of the type code, namely bit 5 (value 32) of each byte, are used to convey chunk properties. This choice means that a human can read off the assigned properties according to whether each letter of the type code is uppercase (bit 5 is 0) or lowercase (bit 5 is 1). However, decoders should test the properties of an unknown chunk by numerically testing the specified bits; testing whether a character is uppercase or lowercase is inefficient, and even incorrect if a locale-specific case definition is used.

It is worth noting that the property bits are an inherent part of the chunk name, and hence are fixed for any chunk type. Thus, TEXT and Text would be unrelated chunk type codes, not the same chunk with different properties. Decoders should recognize type codes by a simple four-byte literal comparison; it is incorrect to perform case conversion on type codes.

The semantics of the property bits are:

Ancillary bit: bit 5 of first byte
0 (uppercase) = critical, 1 (lowercase) = ancillary.

Chunks that are not strictly necessary in order to meaningfully display the contents of the file are known as "ancillary" chunks. A decoder encountering an unknown chunk in which the ancillary bit is 1 can safely ignore the chunk and proceed to display the image. The time chunk (tIME) is an example of an ancillary chunk.

Chunks that are necessary for successful display of the file's contents are called "critical" chunks. A decoder encountering an unknown chunk in which the ancillary bit is 0 must indicate to the user that the image contains information it cannot safely interpret. The image header chunk (IHDR) is an example of a critical chunk.

Private bit: bit 5 of second byte
0 (uppercase) = public, 1 (lowercase) = private.

A public chunk is one that is part of the PNG specification or is registered in the list of PNG special-purpose public chunk types. Applications can also define private (unregistered) chunks for their own purposes. The names of private chunks must have a lowercase second letter, while public chunks will always be assigned names with uppercase second letters. Note that decoders do not need to test the private-chunk property bit, since it has no functional significance; it is simply an administrative convenience to ensure that public and private chunk names will not conflict. See Additional Chunk Types (Section 4.4) and Recommendations for Encoders: Use of private chunks (Section 9.8).

Reserved bit: bit 5 of third byte
Must be 0 (uppercase) always.

The significance of the case of the third letter of the chunk name is reserved for possible future expansion. At the present time all chunk names must have uppercase third letters. (Decoders should not complain about a lowercase third letter, however, as some future version of the PNG specification could define a meaning for this bit. It is sufficient to treat a chunk with a lowercase third letter in the same way as any

other unknown chunk type.)

Safe-to-copy bit: bit 5 of fourth byte
0 (uppercase) = unsafe to copy, 1 (lowercase) = safe to copy.

This property bit is not of interest to pure decoders, but it is needed by PNG editors (programs that modify PNG files).

This bit defines the proper handling of unrecognized chunks in a file that is being modified.

If a chunk's safe-to-copy bit is 1, the chunk may be copied to a modified PNG file whether or not the software recognizes the chunk type, and regardless of the extent of the file modifications.

If a chunk's safe-to-copy bit is 0, it indicates that the chunk depends on the image data. If the program has made any changes to critical chunks, including addition, modification, deletion, or reordering of critical chunks, then unrecognized unsafe chunks must not be copied to the output PNG file. (Of course, if the program does recognize the chunk, it can choose to output an appropriately modified version.)

A PNG editor is always allowed to copy all unrecognized chunks if it has only added, deleted, modified, or reordered ancillary chunks. This implies that it is not permissible for ancillary chunks to depend on other ancillary chunks.

PNG editors that do not recognize a critical chunk must report an error and refuse to process that PNG file at all. The safe/unsafe mechanism is intended for use with ancillary chunks. The safe-to-copy bit will always be 0 for critical chunks.

Rules for PNG editors are discussed further in Chunk Ordering Rules (Chapter 7).

For example, the hypothetical chunk type name "bLOb" has the property bits:

```
bLOb  <-- 32 bit chunk type code represented in text form
||||
|||+- Safe-to-copy bit is 1 (lower case letter; bit 5 is 1)
||+-- Reserved bit is 0     (upper case letter; bit 5 is 0)
|+--- Private bit is 0      (upper case letter; bit 5 is 0)
+---- Ancillary bit is 1    (lower case letter; bit 5 is 1)
```

Therefore, this name represents an ancillary, public, safe-to-copy chunk.

See Rationale: Chunk naming conventions (Section 12.13).

3.4. CRC algorithm

Chunk CRCs are calculated using standard CRC methods with pre and post conditioning, as defined by ISO 3309 [ISO-3309] or ITU-T V.42 [ITU-V42]. The CRC polynomial employed is

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$$

The 32-bit CRC register is initialized to all 1's, and then the data from each byte is processed from the least significant bit (1) to the most significant bit (128). After all the data bytes are processed, the CRC register is inverted (its ones complement is taken). This value is transmitted (stored in the file) MSB first. For the purpose of separating into bytes and ordering, the least significant bit of the 32-bit CRC is defined to be the coefficient of the x^{31} term.

Practical calculation of the CRC always employs a precalculated table to greatly accelerate the computation. See Sample CRC Code (Chapter 15).

4. Chunk Specifications

This chapter defines the standard types of PNG chunks.

4.1. Critical Chunks

All implementations must understand and successfully render the standard critical chunks. A valid PNG image must contain an IHDR chunk, one or more IDAT chunks, and an IEND chunk.

4.1.1. IHDR Image Header

The IHDR chunk must appear FIRST. It contains:

```
Width:              4 bytes
Height:             4 bytes
Bit depth:          1 byte
Color type:         1 byte
Compression method: 1 byte
Filter method:      1 byte
Interlace method:   1 byte
```

Width and height give the image dimensions in pixels. They are 4-byte integers. Zero is an invalid value. The maximum for each is (2^31)-1 in order to accommodate languages that have difficulty with unsigned 4-byte values.

Bit depth is a single-byte integer giving the number of bits per sample (not per pixel, except when a pixel contains just one sample). Valid values are 1, 2, 4, 8, and 16, although not all values are allowed for all color types.

Color type is a single-byte integer that describes the interpretation of the image data. Color type codes represent sums of the following values: 1 (palette used), 2 (color used), and 4 (alpha channel used). Valid values are 0, 2, 3, 4, and 6.

Bit depth restrictions for each color type are imposed to simplify implementations and to prohibit combinations that do not compress well. Decoders must support all legal combinations of bit depth and color type. The allowed combinations are:

Color Type	Allowed Bit Depths	Interpretation
0	1,2,4,8,16	Each pixel is a grayscale sample.
2	8,16	Each pixel is an R,G,B triple.
3	1,2,4,8	Each pixel is a palette index; a PLTE chunk must appear.
4	8,16	Each pixel is a grayscale sample, followed by an alpha sample.
6	8,16	Each pixel is an R,G,B triple, followed by an alpha sample.

Compression method is a single-byte integer that indicates the method used to compress the image data. At present, only compression method 0 (deflate/inflate compression with a 32K sliding window) is defined. All standard PNG images must be compressed with this scheme. The compression method field is provided for possible future expansion or proprietary variants.

Decoders must check this byte and report an error if it holds an unrecognized code. See Deflate/Inflate Compression (Chapter 5) for details.

Filter method is a single-byte integer that indicates the preprocessing method applied to the image data before compression. At present, only filter method 0 (adaptive filtering with five basic filter types) is defined. As with the compression method field, decoders must check this byte and report an error if it holds an unrecognized code. See Filter Algorithms (Chapter 6) for details.

Interlace method is a single-byte integer that indicates the transmission order of the image data. Two values are currently defined: 0 (no interlace) or 1 (Adam7 interlace). See Interlaced data order (Section 2.6) for details.

4.1.2. PLTE Palette

The PLTE chunk contains from 1 to 256 palette entries, each a three-byte series of the form:

```
Red:   1 byte (0 = black, 255 = red)
Green: 1 byte (0 = black, 255 = green)
Blue:  1 byte (0 = black, 255 = blue)
```

The number of entries is determined from the chunk length. A chunk length not divisible by 3 is an error.

This chunk must appear for color type 3, and can appear for color types 2 and 6; it is not allowed for color types 0 and 4. If this chunk does appear, it must precede the first IDAT chunk. There cannot be more than one PLTE chunk.

For color type 3 (indexed color), the PLTE chunk is required. The first entry in PLTE is referenced by pixel value 0, the second by pixel value 1, etc. The number of palette entries must not exceed the range that can be represented in the image bit depth (for example, 2^4 = 16 for a bit depth of 4). It is permissible to have fewer entries than the bit depth would allow. In that case, any out-of-range pixel value found in the image data is an error.

For color types 2 and 6 (truecolor and truecolor with alpha), the PLTE chunk is optional. If present, it provides a suggested set of from 1 to 256 colors to which the truecolor image can be quantized if the viewer cannot display truecolor directly. If PLTE is not present, such a viewer must select colors on its own, but it is often preferable for this to be done once by the encoder. (See Recommendations for Encoders: Suggested palettes, Section 9.5.)

Note that the palette uses 8 bits (1 byte) per sample regardless of the image bit depth specification. In particular, the palette is 8 bits deep even when it is a suggested quantization of a 16-bit truecolor image.

There is no requirement that the palette entries all be used by the image, nor that they all be different.

4.1.3. IDAT Image Data

The IDAT chunk contains the actual image data. To create this

data:

* Begin with image scanlines represented as described in Image layout (Section 2.3); the layout and total size of this raw data are determined by the fields of IHDR.
* Filter the image data according to the filtering method

specified by the IHDR chunk. (Note that with filter method 0, the only one currently defined, this implies prepending a filter type byte to each scanline.)
* Compress the filtered data using the compression method specified by the IHDR chunk.

The IDAT chunk contains the output datastream of the compression algorithm.

To read the image data, reverse this process.

There can be multiple IDAT chunks; if so, they must appear consecutively with no other intervening chunks. The compressed datastream is then the concatenation of the contents of all the IDAT chunks. The encoder can divide the compressed datastream into IDAT chunks however it wishes. (Multiple IDAT chunks are allowed so that encoders can work in a fixed amount of memory; typically the chunk size will correspond to the encoder's buffer size.) It is important to emphasize that IDAT chunk boundaries have no semantic significance and can occur at any point in the compressed datastream. A PNG file in which each IDAT chunk contains only one data byte is legal, though remarkably wasteful of space. (For that matter, zero-length IDAT chunks are legal, though even more wasteful.)

See Filter Algorithms (Chapter 6) and Deflate/Inflate Compression (Chapter 5) for details.

4.1.4. IEND Image Trailer

The IEND chunk must appear LAST. It marks the end of the PNG datastream. The chunk's data field is empty.

4.2. Ancillary Chunks

All ancillary chunks are optional, in the sense that encoders need not write them and decoders can ignore them. However, encoders are encouraged to write the standard ancillary chunks when the information is available, and decoders are encouraged to interpret these chunks when appropriate and feasible.

The standard ancillary chunks are listed in alphabetical order. This is not necessarily the order in which they would appear in a file.

4.2.1. bKGD Background Color

The bKGD chunk specifies a default background color to present the image against. Note that viewers are not bound to honor this chunk; a viewer can choose to use a different background.

For color type 3 (indexed color), the bKGD chunk contains:

Palette index: 1 byte

The value is the palette index of the color to be used as background.

For color types 0 and 4 (grayscale, with or without alpha), bKGD contains:

Gray: 2 bytes, range 0 .. (2^bitdepth)-1

(For consistency, 2 bytes are used regardless of the image bit depth.) The value is the gray level to be used as background.

For color types 2 and 6 (truecolor, with or without alpha), bKGD contains:

```
Red:   2 bytes, range 0 .. (2^bitdepth)-1
Green: 2 bytes, range 0 .. (2^bitdepth)-1
Blue:  2 bytes, range 0 .. (2^bitdepth)-1
```

(For consistency, 2 bytes per sample are used regardless of the image bit depth.) This is the RGB color to be used as background.

When present, the bKGD chunk must precede the first IDAT chunk, and must follow the PLTE chunk, if any.

See Recommendations for Decoders: Background color (Section 10.7).

4.2.2. cHRM Primary Chromaticities and White Point

Applications that need device-independent specification of colors in a PNG file can use the cHRM chunk to specify the 1931 CIE x,y chromaticities of the red, green, and blue primaries used in the image, and the referenced white point. See Color Tutorial (Chapter 14) for more information.

The cHRM chunk contains:

```
White Point x: 4 bytes
White Point y: 4 bytes
Red x:         4 bytes
Red y:         4 bytes
Green x:       4 bytes
Green y:       4 bytes
Blue x:        4 bytes
Blue y:        4 bytes
```

Each value is encoded as a 4-byte unsigned integer, representing the x or y value times 100000. For example, a value of 0.3127 would be stored as the integer 31270.

cHRM is allowed in all PNG files, although it is of little value for grayscale images.

If the encoder does not know the chromaticity values, it should not write a cHRM chunk; the absence of a cHRM chunk indicates that the image's primary colors are device-dependent.

If the cHRM chunk appears, it must precede the first IDAT chunk, and it must also precede the PLTE chunk if present.

See Recommendations for Encoders: Encoder color handling (Section 9.3), and Recommendations for Decoders: Decoder color

handling (Section 10.6).

4.2.3. gAMA Image Gamma

The gAMA chunk specifies the gamma of the camera (or simulated camera) that produced the image, and thus the gamma of the image with respect to the original scene. More precisely, the gAMA chunk encodes the file_gamma value, as defined in Gamma Tutorial (Chapter 13).

The gAMA chunk contains:

    Image gamma: 4 bytes

The value is encoded as a 4-byte unsigned integer, representing gamma times 100000. For example, a gamma of 0.45 would be stored as the integer 45000.

If the encoder does not know the image's gamma value, it should not write a gAMA chunk; the absence of a gAMA chunk indicates that the gamma is unknown.

If the gAMA chunk appears, it must precede the first IDAT chunk, and it must also precede the PLTE chunk if present.

See Gamma correction (Section 2.7), Recommendations for Encoders: Encoder gamma handling (Section 9.2), and Recommendations for Decoders: Decoder gamma handling (Section 10.5).

4.2.4. hIST Image Histogram

The hIST chunk gives the approximate usage frequency of each color in the color palette. A histogram chunk can appear only when a palette chunk appears. If a viewer is unable to provide all the colors listed in the palette, the histogram may help it decide how to choose a subset of the colors for display.

The hIST chunk contains a series of 2-byte (16 bit) unsigned integers. There must be exactly one entry for each entry in

the PLTE chunk. Each entry is proportional to the fraction of pixels in the image that have that palette index; the exact scale factor is chosen by the encoder.

Histogram entries are approximate, with the exception that a zero entry specifies that the corresponding palette entry is not used at all in the image. It is required that a histogram entry be nonzero if there are any pixels of that color.

When the palette is a suggested quantization of a truecolor image, the histogram is necessarily approximate, since a decoder may map pixels to palette entries differently than the encoder did. In this situation, zero entries should not appear.

The hIST chunk, if it appears, must follow the PLTE chunk, and must precede the first IDAT chunk.

See Rationale: Palette histograms (Section 12.14), and Recommendations for Decoders: Suggested-palette and histogram usage (Section 10.10).

4.2.5. pHYs Physical Pixel Dimensions

The pHYs chunk specifies the intended resolution for display of

the image. It contains:

```
Pixels per unit, X axis: 4 bytes (unsigned integer)
Pixels per unit, Y axis: 4 bytes (unsigned integer)
Unit specifier:          1 byte
```

The following values are legal for the unit specifier:

```
0: unit is unknown
1: unit is the meter
```

When the unit specifier is 0, the pHYs chunk defines pixel aspect ratio only; the actual size of the pixels remains unspecified.

Conversion note: one inch is equal to exactly 0.0254 meters.

If this ancillary chunk is not present, pixels are assumed to be square, and the physical size of each pixel is unknown.

If present, this chunk must precede the first IDAT chunk.

See Recommendations for Decoders: Pixel dimensions (Section 10.2).

4.2.6. sBIT Significant Bits

To simplify decoders, PNG specifies that only certain sample bit depths can be used, and further specifies that sample values should be scaled to the full range of possible values at that bit depth. However, the sBIT chunk is provided in order to store the original number of significant bits. This allows decoders to recover the original data losslessly even if the data had a bit depth not directly supported by PNG. We recommend that an encoder emit an sBIT chunk if it has converted the data from a lower bit depth.

For color type 0 (grayscale), the sBIT chunk contains a single byte, indicating the number of bits that were significant in the source data.

For color type 2 (truecolor), the sBIT chunk contains three bytes, indicating the number of bits that were significant in the source data for the red, green, and blue channels, respectively.

For color type 3 (indexed color), the sBIT chunk contains three bytes, indicating the number of bits that were significant in the source data for the red, green, and blue components of the palette entries, respectively.

For color type 4 (grayscale with alpha channel), the sBIT chunk contains two bytes, indicating the number of bits that were significant in the source grayscale data and the source alpha data, respectively.

For color type 6 (truecolor with alpha channel), the sBIT chunk contains four bytes, indicating the number of bits that were significant in the source data for the red, green, blue and alpha channels, respectively.

Each depth specified in sBIT must be greater than zero and less than or equal to the sample depth (which is 8 for indexed-color

images, and the bit depth given in IHDR for other color types).

A decoder need not pay attention to sBIT: the stored image is a valid PNG file of the sample depth indicated by IHDR. However, if the decoder wishes to recover the original data at its original precision, this can be done by right-shifting the stored samples (the stored palette entries, for an indexed-color image). The encoder must scale the data in such a way that the high-order bits match the original data.

If the sBIT chunk appears, it must precede the first IDAT chunk, and it must also precede the PLTE chunk if present.

See Recommendations for Encoders: Bit depth scaling (Section

9.1) and Recommendations for Decoders: Bit depth rescaling (Section 10.4).

4.2.7. tEXt Textual Data

Textual information that the encoder wishes to record with the image can be stored in tEXt chunks. Each tEXt chunk contains a keyword and a text string, in the format:

```
Keyword:        1-79 bytes (character string)
Null separator: 1 byte
Text:           n bytes (character string)
```

The keyword and text string are separated by a zero byte (null character). Neither the keyword nor the text string can contain a null character. Note that the text string is not null-terminated (the length of the chunk is sufficient information to locate the ending). The keyword must be at least one character and less than 80 characters long. The text string can be of any length from zero bytes up to the maximum permissible chunk size less the length of the keyword and separator.

Any number of tEXt chunks can appear, and more than one with the same keyword is permissible.

The keyword indicates the type of information represented by the text string. The following keywords are predefined and should be used where appropriate:

```
Title            Short (one line) title or caption for image
Author           Name of image's creator
Description      Description of image (possibly long)
Copyright        Copyright notice
Creation Time    Time of original image creation
Software         Software used to create the image
Disclaimer       Legal disclaimer
Warning          Warning of nature of content
Source           Device used to create the image
Comment          Miscellaneous comment; conversion from
                 GIF comment
```

For the Creation Time keyword, the date format defined in section 5.2.14 of RFC 1123 is suggested, but not required [RFC-1123]. Decoders should allow for free-format text associated with this or any other keyword.

Other keywords may be invented for other purposes. Keywords of general interest can be registered with the maintainers of the PNG specification. However, it is also permitted to use private unregistered keywords. (Private keywords should be

reasonably self-explanatory, in order to minimize the chance

that the same keyword will be used for incompatible purposes by different people.)

Both keyword and text are interpreted according to the ISO 8859-1 (Latin-1) character set [ISO-8859]. The text string can contain any Latin-1 character. Newlines in the text string should be represented by a single linefeed character (decimal 10); use of other control characters in the text is discouraged.

Keywords must contain only printable Latin-1 characters and spaces; that is, only character codes 32-126 and 161-255 decimal are allowed. To reduce the chances for human misreading of a keyword, leading and trailing spaces are forbidden, as are consecutive spaces. Note also that the non-breaking space (code 160) is not permitted in keywords, since it is visually indistinguishable from an ordinary space.

Keywords must be spelled exactly as registered, so that decoders can use simple literal comparisons when looking for particular keywords. In particular, keywords are considered case-sensitive.

See Recommendations for Encoders: Text chunk processing (Section 9.7) and Recommendations for Decoders: Text chunk processing (Section 10.11).

4.2.8. tIME Image Last-Modification Time

The tIME chunk gives the time of the last image modification (not the time of initial image creation). It contains:

```
Year:   2 bytes (complete; for example, 1995, not 95)
Month:  1 byte (1-12)
Day:    1 byte (1-31)
Hour:   1 byte (0-23)
Minute: 1 byte (0-59)
Second: 1 byte (0-60)    (yes, 60, for leap seconds; not 61,
                          a common error)
```

Universal Time (UTC, also called GMT) should be specified rather than local time.

The tIME chunk is intended for use as an automatically-applied time stamp that is updated whenever the image data is changed. It is recommended that tIME not be changed by PNG editors that do not change the image data. See also the Creation Time tEXt keyword, which can be used for a user-supplied time.

4.2.9. tRNS Transparency

The tRNS chunk specifies that the image uses simple transparency: either alpha values associated with palette

entries (for indexed-color images) or a single transparent color (for grayscale and truecolor images). Although simple transparency is not as elegant as the full alpha channel, it requires less storage space and is sufficient for many common cases.

For color type 3 (indexed color), the tRNS chunk contains a series of one-byte alpha values, corresponding to entries in the PLTE chunk:

```
Alpha for palette index 0:  1 byte
Alpha for palette index 1:  1 byte
... etc ...
```

Each entry indicates that pixels of the corresponding palette index should be treated as having the specified alpha value. Alpha values have the same interpretation as in an 8-bit full alpha channel: 0 is fully transparent, 255 is fully opaque, regardless of image bit depth. The tRNS chunk must not contain more alpha values than there are palette entries, but tRNS can contain fewer values than there are palette entries. In this case, the alpha value for all remaining palette entries is assumed to be 255. In the common case in which only palette index 0 need be made transparent, only a one-byte tRNS chunk is needed.

For color type 0 (grayscale), the tRNS chunk contains a single gray level value, stored in the format:

```
Gray:  2 bytes, range 0 .. (2^bitdepth)-1
```

(For consistency, 2 bytes are used regardless of the image bit depth.) Pixels of the specified gray level are to be treated as transparent (equivalent to alpha value 0); all other pixels are to be treated as fully opaque (alpha value (2^bitdepth)-1).

For color type 2 (truecolor), the tRNS chunk contains a single RGB color value, stored in the format:

```
Red:   2 bytes, range 0 .. (2^bitdepth)-1
Green: 2 bytes, range 0 .. (2^bitdepth)-1
Blue:  2 bytes, range 0 .. (2^bitdepth)-1
```

(For consistency, 2 bytes per sample are used regardless of the image bit depth.) Pixels of the specified color value are to be treated as transparent (equivalent to alpha value 0); all other pixels are to be treated as fully opaque (alpha value (2^bitdepth)-1).

tRNS is prohibited for color types 4 and 6, since a full alpha channel is already present in those cases.

Note: when dealing with 16-bit grayscale or truecolor data, it is important to compare both bytes of the sample values to determine whether a pixel is transparent. Although decoders may drop the low-order byte of the samples for display, this must not occur until after the data has been tested for transparency. For example, if the grayscale level 0x0001 is specified to be transparent, it would be incorrect to compare only the high-order byte and decide that 0x0002 is also transparent.

When present, the tRNS chunk must precede the first IDAT chunk, and must follow the PLTE chunk, if any.

4.2.10. zTXt Compressed Textual Data

The zTXt chunk contains textual data, just as tEXt does; however, zTXt takes advantage of compression. zTXt and tEXt chunks are semantically equivalent, but zTXt is recommended for storing large blocks of text.

A zTXt chunk contains:

```
Keyword:            1-79 bytes (character string)
Null separator:     1 byte
Compression method: 1 byte
Compressed text:    n bytes
```

The keyword and null separator are exactly the same as in the tEXt chunk. Note that the keyword is not compressed. The compression method byte identifies the compression method used in this zTXt chunk. The only value presently defined for it is 0 (deflate/inflate compression). The compression method byte is followed by a compressed datastream that makes up the remainder of the chunk. For compression method 0, this datastream adheres to the zlib datastream format (see Deflate/Inflate Compression, Chapter 5). Decompression of this datastream yields Latin-1 text that is identical to the text that would be stored in an equivalent tEXt chunk.

Any number of zTXt and tEXt chunks can appear in the same file. See the preceding definition of the tEXt chunk for the predefined keywords and the recommended format of the text.

See Recommendations for Encoders: Text chunk processing (Section 9.7), and Recommendations for Decoders: Text chunk processing (Section 10.11).

4.3. Summary of Standard Chunks

This table summarizes some properties of the standard chunk types.

Critical chunks (must appear in this order, except PLTE is optional):

Name	Multiple OK?	Ordering constraints
IHDR	No	Must be first
PLTE	No	Before IDAT
IDAT	Yes	Multiple IDATs must be consecutive
IEND	No	Must be last

Ancillary chunks (need not appear in this order):

Name	Multiple OK?	Ordering constraints
cHRM	No	Before PLTE and IDAT
gAMA	No	Before PLTE and IDAT
sBIT	No	Before PLTE and IDAT
bKGD	No	After PLTE; before IDAT
hIST	No	After PLTE; before IDAT
tRNS	No	After PLTE; before IDAT
pHYs	No	Before IDAT
tIME	No	None
tEXt	Yes	None
zTXt	Yes	None

Standard keywords for tEXt and zTXt chunks:

Title	Short (one line) title or caption for image
Author	Name of image's creator
Description	Description of image (possibly long)
Copyright	Copyright notice
Creation Time	Time of original image creation
Software	Software used to create the image
Disclaimer	Legal disclaimer
Warning	Warning of nature of content
Source	Device used to create the image
Comment	Miscellaneous comment; conversion from GIF comment

4.4. Additional Chunk Types

Additional public PNG chunk types are defined in the document "PNG Special-Purpose Public Chunks" [PNG-EXTENSIONS]. Chunks described there are expected to be less widely supported than those defined in this specification. However, application authors are encouraged to use those chunk types whenever appropriate for their

applications. Additional chunk types can be proposed for inclusion in that list by contacting the PNG specification maintainers at png-info@uunet.uu.net.

New public chunks will only be registered if they are of use to others and do not violate the design philosophy of PNG. Chunk registration is not automatic, although it is the intent of the authors that it be straightforward when a new chunk of potentially wide application is needed. Note that the creation of new critical chunk types is discouraged unless absolutely necessary.

Applications can also use private chunk types to carry data that is not of interest to other applications. See Recommendations for Encoders: Use of private chunks (Section 9.8).

Decoders must be prepared to encounter unrecognized public or private chunk type codes. Unrecognized chunk types must be handled as described in Chunk naming conventions (Section 3.3).

5. Deflate/Inflate Compression

PNG compression method 0 (the only compression method presently defined for PNG) specifies deflate/inflate compression with a 32K sliding window. Deflate compression is an LZ77 derivative used in zip, gzip, pkzip and related programs. Extensive research has been done supporting its patent-free status. Portable C implementations are freely available.

Deflate-compressed datastreams within PNG are stored in the "zlib" format, which has the structure:

```
Compression method/flags code: 1 byte
Additional flags/check bits:   1 byte
Compressed data blocks:        n bytes
Check value:                   4 bytes
```

Further details on this format are given in the zlib specification [RFC-1950].

For PNG compression method 0, the zlib compression method/flags code must specify method code 8 ("deflate" compression) and an LZ77 window size of not more than 32K. Note that the zlib compression method number is not the same as the PNG compression method number. The

additional flags must not specify a preset dictionary.

The compressed data within the zlib datastream is stored as a series of blocks, each of which can represent raw (uncompressed) data, LZ77-compressed data encoded with fixed Huffman codes, or LZ77-compressed data encoded with custom Huffman codes. A marker bit in the final block identifies it as the last block, allowing the decoder to recognize the end of the compressed datastream. Further details on the compression algorithm and the encoding are given in the

deflate specification [RFC-1951].

The check value stored at the end of the zlib datastream is calculated on the uncompressed data represented by the datastream. Note that the algorithm used is not the same as the CRC calculation used for PNG chunk check values. The zlib check value is useful mainly as a cross-check that the deflate and inflate algorithms are implemented correctly. Verifying the chunk CRCs provides adequate confidence that the PNG file has been transmitted undamaged.

In a PNG file, the concatenation of the contents of all the IDAT chunks makes up a zlib datastream as specified above. This datastream decompresses to filtered image data as described elsewhere in this document.

It is important to emphasize that the boundaries between IDAT chunks are arbitrary and can fall anywhere in the zlib datastream. There is not necessarily any correlation between IDAT chunk boundaries and deflate block boundaries or any other feature of the zlib data. For example, it is entirely possible for the terminating zlib check value to be split across IDAT chunks.

In the same vein, there is no required correlation between the structure of the image data (i.e., scanline boundaries) and deflate block boundaries or IDAT chunk boundaries. The complete image data is represented by a single zlib datastream that is stored in some number of IDAT chunks; a decoder that assumes any more than this is incorrect. (Of course, some encoder implementations may emit files in which some of these structures are indeed related. But decoders cannot rely on this.)

PNG also uses zlib datastreams in zTXt chunks. In a zTXt chunk, the remainder of the chunk following the compression method byte is a zlib datastream as specified above. This datastream decompresses to the user-readable text described by the chunk's keyword. Unlike the image data, such datastreams are not split across chunks; each zTXt chunk contains an independent zlib datastream.

Additional documentation and portable C code for deflate and inflate are available from the Info-ZIP archives at <URL:ftp://ftp.uu.net/pub/archiving/zip/>.

6. Filter Algorithms

This chapter describes the filter algorithms that can be applied before compression. The purpose of these filters is to prepare the image data for optimum compression.

PNG filter method 0 defines five basic filter types:

Type	Name
0	None
1	Sub
2	Up
3	Average
4	Paeth

(Note that filter method 0 in IHDR specifies exactly this set of five filter types. If the set of filter types is ever extended, a different filter method number will be assigned to the extended set, so that decoders need not decompress the data to discover that it contains unsupported filter types.)

The encoder can choose which of these filter algorithms to apply on a scanline-by-scanline basis. In the image data sent to the compression step, each scanline is preceded by a filter type byte that specifies the filter algorithm used for that scanline.

Filtering algorithms are applied to bytes, not to pixels, regardless of the bit depth or color type of the image. The filtering algorithms work on the byte sequence formed by a scanline that has been represented as described in Image layout (Section 2.3). If the image includes an alpha channel, the alpha data is filtered in the same way as the image data.

When the image is interlaced, each pass of the interlace pattern is treated as an independent image for filtering purposes. The filters work on the byte sequences formed by the pixels actually transmitted during a pass, and the "previous scanline" is the one previously transmitted in the same pass, not the one adjacent in the complete image. Note that the subimage transmitted in any one pass is always rectangular, but is of smaller width and/or height than the complete image. Filtering is not applied when this subimage is empty.

For all filters, the bytes "to the left of" the first pixel in a scanline must be treated as being zero. For filters that refer to the prior scanline, the entire prior scanline must be treated as being zeroes for the first scanline of an image (or of a pass of an interlaced image).

To reverse the effect of a filter, the decoder must use the decoded values of the prior pixel on the same line, the pixel immediately above the current pixel on the prior line, and the pixel just to the

left of the pixel above. This implies that at least one scanline's worth of image data must be stored by the decoder at all times. Even though some filter types do not refer to the prior scanline, the decoder must always store each scanline as it is decoded, since the next scanline might use a filter that refers to it.

PNG imposes no restriction on which filter types can be applied to an image. However, the filters are not equally effective on all types of data. See Recommendations for Encoders: Filter selection (Section

9.6).

See also Rationale: Filtering (Section 12.9).

6.1. Filter type 0: None

With the None filter, the scanline is transmitted unmodified; it is only necessary to insert a filter type byte before the data.

6.2. Filter type 1: Sub

The Sub filter transmits the difference between each byte and the value of the corresponding byte of the prior pixel.

To compute the Sub filter, apply the following formula to each byte of the scanline:

```
Sub(x) = Raw(x) - Raw(x-bpp)
```

where x ranges from zero to the number of bytes representing the scanline minus one, Raw(x) refers to the raw data byte at that byte position in the scanline, and bpp is defined as the number of bytes per complete pixel, rounding up to one. For example, for color type 2 with a bit depth of 16, bpp is equal to 6 (three samples, two bytes per sample); for color type 0 with a bit depth of 2, bpp is equal to 1 (rounding up); for color type 4 with a bit depth of 16, bpp is equal to 4 (two-byte grayscale sample, plus two-byte alpha sample).

Note this computation is done for each byte, regardless of bit depth. In a 16-bit image, each MSB is predicted from the preceding MSB and each LSB from the preceding LSB, because of the way that bpp is defined.

Unsigned arithmetic modulo 256 is used, so that both the inputs and outputs fit into bytes. The sequence of Sub values is transmitted as the filtered scanline.

For all x < 0, assume Raw(x) = 0.

To reverse the effect of the Sub filter after decompression, output the following value:


```
Sub(x) + Raw(x-bpp)
```

(computed mod 256), where Raw refers to the bytes already decoded.

6.3. Filter type 2: Up

The Up filter is just like the Sub filter except that the pixel immediately above the current pixel, rather than just to its left, is used as the predictor.

To compute the Up filter, apply the following formula to each byte of the scanline:

```
Up(x) = Raw(x) - Prior(x)
```

where x ranges from zero to the number of bytes representing the scanline minus one, Raw(x) refers to the raw data byte at that byte position in the scanline, and Prior(x) refers to the unfiltered bytes of the prior scanline.

Note this is done for each byte, regardless of bit depth. Unsigned arithmetic modulo 256 is used, so that both the inputs

and outputs fit into bytes. The sequence of Up values is transmitted as the filtered scanline.

On the first scanline of an image (or of a pass of an interlaced image), assume Prior(x) = 0 for all x.

To reverse the effect of the Up filter after decompression, output the following value:

```
Up(x) + Prior(x)
```

(computed mod 256), where Prior refers to the decoded bytes of the prior scanline.

6.4. Filter type 3: Average

The Average filter uses the average of the two neighboring pixels (left and above) to predict the value of a pixel.

To compute the Average filter, apply the following formula to each byte of the scanline:

```
Average(x) = Raw(x) - floor((Raw(x-bpp)+Prior(x))/2)
```

where x ranges from zero to the number of bytes representing the scanline minus one, Raw(x) refers to the raw data byte at that byte position in the scanline, Prior(x) refers to the unfiltered bytes of the prior scanline, and bpp is defined as for the Sub filter.

Note this is done for each byte, regardless of bit depth. The sequence of Average values is transmitted as the filtered scanline.

The subtraction of the predicted value from the raw byte must be done modulo 256, so that both the inputs and outputs fit into bytes. However, the sum Raw(x-bpp)+Prior(x) must be formed without overflow (using at least nine-bit arithmetic). floor() indicates that the result of the division is rounded to the next lower integer if fractional; in other words, it is an integer division or right shift operation.

For all x < 0, assume Raw(x) = 0. On the first scanline of an image (or of a pass of an interlaced image), assume Prior(x) = 0 for all x.

To reverse the effect of the Average filter after decompression, output the following value:

```
Average(x) + floor((Raw(x-bpp)+Prior(x))/2)
```

where the result is computed mod 256, but the prediction is calculated in the same way as for encoding. Raw refers to the bytes already decoded, and Prior refers to the decoded bytes of the prior scanline.

6.5. Filter type 4: Paeth

The Paeth filter computes a simple linear function of the three neighboring pixels (left, above, upper left), then chooses as predictor the neighboring pixel closest to the computed value. This technique is due to Alan W. Paeth [PAETH].

To compute the Paeth filter, apply the following formula to each byte of the scanline:

```
Paeth(x) = Raw(x) - PaethPredictor(Raw(x-bpp), Prior(x),
                                   Prior(x-bpp))
```

where x ranges from zero to the number of bytes representing the scanline minus one, Raw(x) refers to the raw data byte at that byte position in the scanline, Prior(x) refers to the unfiltered bytes of the prior scanline, and bpp is defined as for the Sub filter.

Note this is done for each byte, regardless of bit depth. Unsigned arithmetic modulo 256 is used, so that both the inputs and outputs fit into bytes. The sequence of Paeth values is transmitted as the filtered scanline.

The PaethPredictor function is defined by the following pseudocode:


```
function PaethPredictor (a, b, c)
begin
     ; a = left, b = above, c = upper left
     p := a + b - c        ; initial estimate
     pa := abs(p - a)      ; distances to a, b, c
     pb := abs(p - b)
     pc := abs(p - c)
     ; return nearest of a,b,c,
     ; breaking ties in order a,b,c.
     if pa <= pb AND pa <= pc then return a
     else if pb <= pc then return b
     else return c
end
```

The calculations within the PaethPredictor function must be performed exactly, without overflow. Arithmetic modulo 256 is to be used only for the final step of subtracting the function result from the target byte value.

Note that the order in which ties are broken is critical and must not be altered. The tie break order is: pixel to the left, pixel above, pixel to the upper left. (This order differs from that given in Paeth's article.)

For all x < 0, assume Raw(x) = 0 and Prior(x) = 0. On the first scanline of an image (or of a pass of an interlaced image), assume Prior(x) = 0 for all x.

To reverse the effect of the Paeth filter after decompression, output the following value:

```
Paeth(x) + PaethPredictor(Raw(x-bpp), Prior(x), Prior(x-bpp))
```

(computed mod 256), where Raw and Prior refer to bytes already decoded. Exactly the same PaethPredictor function is used by both encoder and decoder.

7. Chunk Ordering Rules

To allow new chunk types to be added to PNG, it is necessary to establish rules about the ordering requirements for all chunk types. Otherwise a PNG editing program cannot know what to do when it encounters an unknown chunk.

We define a "PNG editor" as a program that modifies a PNG file and wishes to preserve as much as possible of the ancillary information in the file. Two examples of PNG editors are a program that adds or modifies text chunks, and a program that adds a suggested palette to

a truecolor PNG file. Ordinary image editors are not PNG editors in this sense, because they usually discard all unrecognized information while reading in an image. (Note: we strongly encourage programs handling PNG files to preserve ancillary information whenever

possible.)

As an example of possible problems, consider a hypothetical new ancillary chunk type that is safe-to-copy and is required to appear after PLTE if PLTE is present. If our program to add a suggested PLTE does not recognize this new chunk, it may insert PLTE in the wrong place, namely after the new chunk. We could prevent such problems by requiring PNG editors to discard all unknown chunks, but that is a very unattractive solution. Instead, PNG requires ancillary chunks not to have ordering restrictions like this.

To prevent this type of problem while allowing for future extension, we put some constraints on both the behavior of PNG editors and the allowed ordering requirements for chunks.

7.1. Behavior of PNG editors

The rules for PNG editors are:

* When copying an unknown unsafe-to-copy ancillary chunk, a PNG editor must not move the chunk relative to any critical chunk. It can relocate the chunk freely relative to other ancillary chunks that occur between the same pair of critical chunks. (This is well defined since the editor must not add, delete, modify, or reorder critical chunks if it is preserving unknown unsafe-to-copy chunks.)
* When copying an unknown safe-to-copy ancillary chunk, a PNG editor must not move the chunk from before IDAT to after IDAT or vice versa. (This is well defined because IDAT is always present.) Any other reordering is permitted.
* When copying a known ancillary chunk type, an editor need only honor the specific chunk ordering rules that exist for that chunk type. However, it can always choose to apply the above general rules instead.
* PNG editors must give up on encountering an unknown critical chunk type, because there is no way to be certain that a valid file will result from modifying a file containing such a chunk. (Note that simply discarding the chunk is not good enough, because it might have unknown implications for the interpretation of other chunks.)

These rules are expressed in terms of copying chunks from an input file to an output file, but they apply in the obvious way if a PNG file is modified in place.

See also Chunk naming conventions (Section 3.3).

7.2. Ordering of ancillary chunks

The ordering rules for an ancillary chunk type cannot be any stricter than this:

* Unsafe-to-copy chunks can have ordering requirements

relative to critical chunks.
* Safe-to-copy chunks can have ordering requirements relative to IDAT.

The actual ordering rules for any particular ancillary chunk type may be weaker. See for example the ordering rules for the standard ancillary chunk types (Summary of Standard Chunks, Section 4.3).

Decoders must not assume more about the positioning of any ancillary chunk than is specified by the chunk ordering rules. In particular, it is never valid to assume that a specific ancillary chunk type occurs with any particular positioning relative to other ancillary chunks. (For example, it is unsafe to assume that your private ancillary chunk occurs immediately before IEND. Even if your application always writes it there, a PNG editor might have inserted some other ancillary chunk after it. But you can safely assume that your chunk will remain somewhere between IDAT and IEND.)

7.3. Ordering of critical chunks

Critical chunks can have arbitrary ordering requirements, because PNG editors are required to give up if they encounter unknown critical chunks. For example, IHDR has the special ordering rule that it must always appear first. A PNG editor, or indeed any PNG-writing program, must know and follow the ordering rules for any critical chunk type that it can emit.

8. Miscellaneous Topics

8.1. File name extension

On systems where file names customarily include an extension signifying file type, the extension ".png" is recommended for PNG files. Lower case ".png" is preferred if file names are case-sensitive.

8.2. Internet media type

The PNG authors intend to register "image/png" as the Internet Media Type for PNG [RFC-1521, RFC-1590]. At the date of this document, the media type registration process had not been completed. It is recommended that implementations also recognize the interim media type "image/x-png".

8.3. Macintosh file layout

In the Apple Macintosh system, the following conventions are recommended:

* The four-byte file type code for PNG files is "PNGf". (This code has been registered with Apple for PNG files.) The creator code will vary depending on the creating application.
* The contents of the data fork shall be a PNG file exactly as described in the rest of this specification.
* The contents of the resource fork are unspecified. It may be empty or may contain application-dependent resources.
* When transferring a Macintosh PNG file to a non-Macintosh system, only the data fork should be transferred.

8.4. Multiple-image extension

PNG itself is strictly a single-image format. However, it may be

necessary to store multiple images within one file; for example, this is needed to convert some GIF files. In the future, a multiple-image format based on PNG may be defined. Such a format will be considered a separate file format and will have a different signature. PNG-supporting applications may or may not choose to support the multiple-image format.

See Rationale: Why not these features? (Section 12.3).

8.5. Security considerations

A PNG file or datastream is composed of a collection of explicitly typed "chunks". Chunks whose contents are defined by the specification could actually contain anything, including malicious code. But there is no known risk that such malicious code could be executed on the recipient's computer as a result of decoding the PNG image.

The possible security risks associated with future chunk types cannot be specified at this time. Security issues will be considered when evaluating chunks proposed for registration as public chunks. There is no additional security risk associated with unknown or unimplemented chunk types, because such chunks will be ignored, or at most be copied into another PNG file.

The tEXt and zTXt chunks contain data that is meant to be displayed as plain text. It is possible that if the decoder displays such text without filtering out control characters, especially the ESC (escape) character, certain systems or terminals could behave in undesirable and insecure ways. We recommend that decoders filter out control characters to avoid this risk; see Recommendations for Decoders: Text chunk processing (Section 10.11).

Because every chunk's length is available at its beginning, and because every chunk has a CRC trailer, there is a very robust defense against corrupted data and against fraudulent chunks that attempt to overflow the decoder's buffers. Also, the PNG

signature bytes provide early detection of common file transmission errors.

A decoder that fails to check CRCs could be subject to data corruption. The only likely consequence of such corruption is incorrectly displayed pixels within the image. Worse things might happen if the CRC of the IHDR chunk is not checked and the width or height fields are corrupted. See Recommendations for Decoders: Error checking (Section 10.1).

A poorly written decoder might be subject to buffer overflow, because chunks can be extremely large, up to (2^31)-1 bytes long. But properly written decoders will handle large chunks without difficulty.

9. Recommendations for Encoders

This chapter gives some recommendations for encoder behavior. The only absolute requirement on a PNG encoder is that it produce files that conform to the format specified in the preceding chapters. However, best results will usually be achieved by following these recommendations.

9.1. Bit depth scaling

When encoding input samples that have a bit depth that cannot be directly represented in PNG, the encoder must scale the samples up

to a bit depth that is allowed by PNG. The most accurate scaling method is the linear equation

```
output = ROUND(input * MAXOUTSAMPLE / MAXINSAMPLE)
```

where the input samples range from 0 to MAXINSAMPLE and the outputs range from 0 to MAXOUTSAMPLE (which is (2^bitdepth)-1).

A close approximation to the linear scaling method can be achieved by "left bit replication", which is shifting the valid bits to begin in the most significant bit and repeating the most significant bits into the open bits. This method is often faster to compute than linear scaling. As an example, assume that 5-bit samples are being scaled up to 8 bits. If the source sample value is 27 (in the range from 0-31), then the original bits are:

```
4 3 2 1 0
---------
1 1 0 1 1
```

Left bit replication gives a value of 222:


```
7 6 5 4 3  2 1 0
----------------
1 1 0 1 1  1 1 0
|=======|  |===|
    |      Leftmost Bits Repeated to Fill Open Bits
    |
Original Bits
```

which matches the value computed by the linear equation. Left bit replication usually gives the same value as linear scaling, and is never off by more than one.

A distinctly less accurate approximation is obtained by simply left-shifting the input value and filling the low order bits with zeroes. This scheme cannot reproduce white exactly, since it does not generate an all-ones maximum value; the net effect is to darken the image slightly. This method is not recommended in general, but it does have the effect of improving compression, particularly when dealing with greater-than-eight-bit sample depths. Since the relative error introduced by zero-fill scaling is small at high bit depths, some encoders may choose to use it. Zero-fill should not be used for alpha channel data, however, since many decoders will special-case alpha values of all zeroes and all ones. It is important to represent both those values exactly in the scaled data.

When the encoder writes an sBIT chunk, it is required to do the scaling in such a way that the high-order bits of the stored samples match the original data. That is, if the sBIT chunk specifies a bit depth of S, the high-order S bits of the stored data must agree with the original S-bit data values. This allows decoders to recover the original data by shifting right. The added low-order bits are not constrained. Note that all the above scaling methods meet this restriction.

When scaling up source data, it is recommended that the low-order bits be filled consistently for all samples; that is, the same source value should generate the same sample value at any pixel position. This improves compression by reducing the number of distinct sample values. However, this is not a requirement, and

some encoders may choose not to follow it. For example, an encoder might instead dither the low-order bits, improving displayed image quality at the price of increasing file size.

In some applications the original source data may have a range that is not a power of 2. The linear scaling equation still works for this case, although the shifting methods do not. It is recommended that an sBIT chunk not be written for such images, since sBIT suggests that the original data range was exactly 0..2^S-1.

9.2. Encoder gamma handling

See Gamma Tutorial (Chapter 13) if you aren't already familiar with gamma issues.

Proper handling of gamma encoding and the gAMA chunk in an encoder depends on the prior history of the sample values and on whether these values have already been quantized to integers.

If the encoder has access to sample intensity values in floating-point or high-precision integer form (perhaps from a computer image renderer), then it is recommended that the encoder perform its own gamma encoding before quantizing the data to integer values for storage in the file. Applying gamma encoding at this stage results in images with fewer banding artifacts at a given sample bit depth, or allows smaller samples while retaining the same visual quality.

A linear intensity level, expressed as a floating-point value in the range 0 to 1, can be converted to a gamma-encoded sample value by

```
sample = ROUND((intensity ^ encoder_gamma) * MAXSAMPLEVAL)
```

The file_gamma value to be written in the PNG gAMA chunk is the same as encoder_gamma in this equation, since we are assuming the initial intensity value is linear (in effect, camera_gamma is 1.0).

If the image is being written to a file only, the encoder_gamma value can be selected somewhat arbitrarily. Values of 0.45 or 0.5 are generally good choices because they are common in video systems, and so most PNG decoders should do a good job displaying such images.

Some image renderers may simultaneously write the image to a PNG file and display it on-screen. The displayed pixels should be gamma corrected for the display system and viewing conditions in use, so that the user sees a proper representation of the intended scene. An appropriate gamma correction value is

```
screen_gc = viewing_gamma / display_gamma
```

If the renderer wants to write the same gamma-corrected sample values to the PNG file, avoiding a separate gamma-encoding step for file output, then this screen_gc value should be written in the gAMA chunk. This will allow a PNG decoder to reproduce what the file's originator saw on screen during rendering (provided the decoder properly supports arbitrary values in a gAMA chunk).

However, it is equally reasonable for a renderer to apply gamma correction for screen display using a gamma appropriate to the

viewing conditions, and to separately gamma-encode the sample values for file storage using a standard value of gamma such as 0.5. In fact, this is preferable, since some PNG decoders may not accurately display images with unusual gAMA values.

Computer graphics renderers often do not perform gamma encoding, instead making sample values directly proportional to scene light intensity. If the PNG encoder receives sample values that have already been quantized into linear-light integer values, there is no point in doing gamma encoding on them; that would just result in further loss of information. The encoder should just write the sample values to the PNG file. This "linear" sample encoding is equivalent to gamma encoding with a gamma of 1.0, so graphics programs that produce linear samples should always emit a gAMA chunk specifying a gamma of 1.0.

When the sample values come directly from a piece of hardware, the correct gAMA value is determined by the gamma characteristic of the hardware. In the case of video digitizers ("frame grabbers"), gAMA should be 0.45 or 0.5 for NTSC (possibly less for PAL or SECAM) since video camera transfer functions are standardized. Image scanners are less predictable. Their output samples may be linear (gamma 1.0) since CCD sensors themselves are linear, or the scanner hardware may have already applied gamma correction designed to compensate for dot gain in subsequent printing (gamma of about 0.57), or the scanner may have corrected the samples for display on a CRT (gamma of 0.4-0.5). You will need to refer to the scanner's manual, or even scan a calibrated gray wedge, to determine what a particular scanner does.

File format converters generally should not attempt to convert supplied images to a different gamma. Store the data in the PNG file without conversion, and record the source gamma if it is known. Gamma alteration at file conversion time causes requantization of the set of intensity levels that are represented, introducing further roundoff error with little benefit. It's almost always better to just copy the sample values intact from the input to the output file.

In some cases, the supplied image may be in an image format (e.g., TIFF) that can describe the gamma characteristic of the image. In such cases, a file format converter is strongly encouraged to write a PNG gAMA chunk that corresponds to the known gamma of the source image. Note that some file formats specify the gamma of the display system, not the camera. If the input file's gamma value is greater than 1.0, it is almost certainly a display system gamma, and you should use its reciprocal for the PNG gAMA.

If the encoder or file format converter does not know how an image was originally created, but does know that the image has been displayed satisfactorily on a display with gamma display_gamma under lighting conditions where a particular viewing_gamma is

appropriate, then the image can be marked as having the file_gamma:

```
file_gamma = viewing_gamma / display_gamma
```

This will allow viewers of the PNG file to see the same image that

the person running the file format converter saw. Although this may not be precisely the correct value of the image gamma, it's better to write a gAMA chunk with an approximately right value than to omit the chunk and force PNG decoders to guess at an appropriate gamma.

On the other hand, if the image file is being converted as part of a "bulk" conversion, with no one looking at each image, then it is better to omit the gAMA chunk entirely. If the image gamma must be guessed at, leave it to the decoder to do the guessing.

Gamma does not apply to alpha samples; alpha is always represented linearly.

See also Recommendations for Decoders: Decoder gamma handling (Section 10.5).

9.3. Encoder color handling

See Color Tutorial (Chapter 14) if you aren't already familiar with color issues.

If it is possible for the encoder to determine the chromaticities of the source display primaries, or to make a strong guess based on the origin of the image or the hardware running it, then the encoder is strongly encouraged to output the cHRM chunk. If it does so, the gAMA chunk should also be written; decoders can do little with cHRM if gAMA is missing.

Video created with recent video equipment probably uses the CCIR 709 primaries and D65 white point [ITU-BT709], which are:

	R	G	B	White
x	0.640	0.300	0.150	0.3127
y	0.330	0.600	0.060	0.3290

An older but still very popular video standard is SMPTE-C [SMPTE-170M]:

	R	G	B	White
x	0.630	0.310	0.155	0.3127
y	0.340	0.595	0.070	0.3290

The original NTSC color primaries have not been used in decades. Although you may still find the NTSC numbers listed in standards documents, you won't find any images that actually use them.

Scanners that produce PNG files as output should insert the filter chromaticities into a cHRM chunk and the camera_gamma into a gAMA chunk.

In the case of hand-drawn or digitally edited images, you have to determine what monitor they were viewed on when being produced. Many image editing programs allow you to specify what type of monitor you are using. This is often because they are working in some device-independent space internally. Such programs have enough information to write valid cHRM and gAMA chunks, and should do so automatically.

If the encoder is compiled as a portion of a computer image renderer that performs full-spectral rendering, the monitor values that were used to convert from the internal device-independent color space to RGB should be written into the cHRM chunk. Any colors that are outside the gamut of the chosen RGB device should be clipped or otherwise constrained to be within the gamut; PNG does not store out of gamut colors.

If the computer image renderer performs calculations directly in device-dependent RGB space, a cHRM chunk should not be written unless the scene description and rendering parameters have been adjusted to look good on a particular monitor. In that case, the data for that monitor (if known) should be used to construct a cHRM chunk.

There are often cases where an image's exact origins are unknown, particularly if it began life in some other format. A few image formats store calibration information, which can be used to fill in the cHRM chunk. For example, all PhotoCD images use the CCIR 709 primaries and D65 whitepoint, so these values can be written into the cHRM chunk when converting a PhotoCD file. PhotoCD also uses the SMPTE-170M transfer function, which is closely approximated by a gAMA of 0.5. (PhotoCD can store colors outside the RGB gamut, so the image data will require gamut mapping before writing to PNG format.) TIFF 6.0 files can optionally store calibration information, which if present should be used to construct the cHRM chunk. GIF and most other formats do not store any calibration information.

It is not recommended that file format converters attempt to convert supplied images to a different RGB color space. Store the data in the PNG file without conversion, and record the source primary chromaticities if they are known. Color space transformation at file conversion time is a bad idea because of gamut mismatches and rounding errors. As with gamma conversions, it's better to store the data losslessly and incur at most one conversion when the image is finally displayed.

See also Recommendations for Decoders: Decoder color handling (Section 10.6).

9.4. Alpha channel creation

The alpha channel can be regarded either as a mask that temporarily hides transparent parts of the image, or as a means for constructing a non-rectangular image. In the first case, the color values of fully transparent pixels should be preserved for future use. In the second case, the transparent pixels carry no useful data and are simply there to fill out the rectangular image area required by PNG. In this case, fully transparent pixels should all be assigned the same color value for best compression.

Encoders should keep in mind the possibility that a decoder will ignore transparency control. Hence, the colors assigned to transparent pixels should be reasonable background colors whenever feasible.

For applications that do not require a full alpha channel, or cannot afford the price in compression efficiency, the tRNS transparency chunk is also available.

If the image has a known background color, this color should be written in the bKGD chunk. Even decoders that ignore transparency may use the bKGD color to fill unused screen area.

If the original image has premultiplied (also called "associated") alpha data, convert it to PNG's non-premultiplied format by dividing each sample value by the corresponding alpha value, then multiplying by the maximum value for the image bit depth, and rounding to the nearest integer. In valid premultiplied data, the sample values never exceed their corresponding alpha values, so the result of the division should always be in the range 0 to 1. If the alpha value is zero, output black (zeroes).

9.5. Suggested palettes

A PLTE chunk can appear in truecolor PNG files. In such files, the chunk is not an essential part of the image data, but simply represents a suggested palette that viewers may use to present the image on indexed-color display hardware. A suggested palette is of no interest to viewers running on truecolor hardware.

If an encoder chooses to provide a suggested palette, it is recommended that a hIST chunk also be written to indicate the relative importance of the palette entries. The histogram values are most easily computed as "nearest neighbor" counts, that is, the approximate usage of each palette entry if no dithering is applied. (These counts will often be available for free as a consequence of developing the suggested palette.)

For images of color type 2 (truecolor without alpha channel), it is recommended that the palette and histogram be computed with reference to the RGB data only, ignoring any transparent-color

specification. If the file uses transparency (has a tRNS chunk), viewers can easily adapt the resulting palette for use with their intended background color. They need only replace the palette entry closest to the tRNS color with their background color (which may or may not match the file's bKGD color, if any).

For images of color type 6 (truecolor with alpha channel), it is recommended that a bKGD chunk appear and that the palette and histogram be computed with reference to the image as it would appear after compositing against the specified background color. This definition is necessary to ensure that useful palette entries are generated for pixels having fractional alpha values. The resulting palette will probably only be useful to viewers that present the image against the same background color. It is recommended that PNG editors delete or recompute the palette if they alter or remove the bKGD chunk in an image of color type 6. If PLTE appears without bKGD in an image of color type 6, the circumstances under which the palette was computed are unspecified.

9.6. Filter selection

For images of color type 3 (indexed color), filter type 0 (none) is usually the most effective.

Filter type 0 is also recommended for images of bit depths less than 8. For low-bit-depth grayscale images, it may be a net win to expand the image to 8-bit representation and apply filtering, but this is rare.

For truecolor and grayscale images, any of the five filters may prove the most effective. If an encoder uses a fixed filter, the Paeth filter is most likely to be the best.

For best compression of truecolor and grayscale images, we recommend an adaptive filtering approach in which a filter is chosen for each scanline. The following simple heuristic has performed well in early tests: compute the output scanline using all five filters, and select the filter that gives the smallest sum of absolute values of outputs. (Consider the output bytes as signed differences for this test.) This method usually outperforms any single fixed filter choice. However, it is likely that much better heuristics will be found as more experience is gained with PNG.

Filtering according to these recommendations is effective on interlaced as well as noninterlaced images.

9.7. Text chunk processing

A nonempty keyword must be provided for each text chunk. The generic keyword "Comment" can be used if no better description of

the text is available. If a user-supplied keyword is used, be sure to check that it meets the restrictions on keywords.

PNG text strings are expected to use the Latin-1 character set. Encoders should avoid storing characters that are not defined in Latin-1, and should provide character code remapping if the local system's character set is not Latin-1.

Encoders should discourage the creation of single lines of text longer than 79 characters, in order to facilitate easy reading.

It is recommended that text items less than 1K (1024 bytes) in size be output using uncompressed tEXt chunks. In particular, it is recommended that the basic title and author keywords always be output using uncompressed tEXt chunks. Lengthy disclaimers, on the other hand, are ideal candidates for zTXt.

Placing large tEXt and zTXt chunks after the image data (after IDAT) can speed up image display in some situations, since the decoder won't have to read over the text to get to the image data. But it is recommended that small text chunks, such as the image title, appear before IDAT.

9.8. Use of private chunks

Applications can use PNG private chunks to carry information that need not be understood by other applications. Such chunks must be given names with lowercase second letters, to ensure that they can never conflict with any future public chunk definition. Note, however, that there is no guarantee that some other application will not use the same private chunk name. If you use a private chunk type, it is prudent to store additional identifying information at the beginning of the chunk data.

Use an ancillary chunk type (lowercase first letter), not a critical chunk type, for all private chunks that store information that is not absolutely essential to view the image. Creation of private critical chunks is discouraged because they render PNG files unportable. Such chunks should not be used in publicly available software or files. If private critical chunks are essential for your application, it is recommended that one appear near the start of the file, so that a standard decoder need not read very far before discovering that it cannot handle the file.

If you want others outside your organization to understand a chunk type that you invent, contact the maintainers of the PNG specification to submit a proposed chunk name and definition for addition to the list of special-purpose public chunks (see Additional Chunk Types, Section 4.4). Note that a proposed public chunk name (with uppercase second letter) must not be used in publicly available software or files until registration has been approved.

If an ancillary chunk contains textual information that might be of interest to a human user, you should not create a special chunk type for it. Instead use a tEXt chunk and define a suitable keyword. That way, the information will be available to users not using your software.

Keywords in tEXt chunks should be reasonably self-explanatory, since the idea is to let other users figure out what the chunk contains. If of general usefulness, new keywords can be registered with the maintainers of the PNG specification. But it is permissible to use keywords without registering them first.

9.9. Private type and method codes

This specification defines the meaning of only some of the possible values of some fields. For example, only compression method 0 and filter types 0 through 4 are defined. Use numbers greater than 127 when inventing experimental or private definitions of values for any of these fields. Numbers below 128 are reserved for possible future public extensions of this specification. Note that use of private type codes may render a file unreadable by standard decoders. Such codes are strongly discouraged except for experimental purposes, and should not appear in publicly available software or files.

10. Recommendations for Decoders

This chapter gives some recommendations for decoder behavior. The only absolute requirement on a PNG decoder is that it successfully read any file conforming to the format specified in the preceding chapters. However, best results will usually be achieved by following these recommendations.

10.1. Error checking

To ensure early detection of common file-transfer problems, decoders should verify that all eight bytes of the PNG file signature are correct. (See Rationale: PNG file signature, Section 12.11.) A decoder can have additional confidence in the file's integrity if the next eight bytes are an IHDR chunk header with the correct chunk length.

Unknown chunk types must be handled as described in Chunk naming conventions (Section 3.3). An unknown chunk type is not to be treated as an error unless it is a critical chunk.

It is strongly recommended that decoders verify the CRC on each chunk.

In some situations it is desirable to check chunk headers (length and type code) before reading the chunk data and CRC. The chunk type can be checked for plausibility by seeing whether all four

bytes are ASCII letters (codes 65-90 and 97-122); note that this need only be done for unrecognized type codes. If the total file size is known (from file system information, HTTP protocol, etc), the chunk length can be checked for plausibility as well.

If CRCs are not checked, dropped/added data bytes or an erroneous chunk length can cause the decoder to get out of step and misinterpret subsequent data as a chunk header. Verifying that the chunk type contains letters is an inexpensive way of providing early error detection in this situation.

For known-length chunks such as IHDR, decoders should treat an unexpected chunk length as an error. Future extensions to this specification will not add new fields to existing chunks; instead, new chunk types will be added to carry new information.

Unexpected values in fields of known chunks (for example, an unexpected compression method in the IHDR chunk) must be checked for and treated as errors. However, it is recommended that unexpected field values be treated as fatal errors only in critical chunks. An unexpected value in an ancillary chunk can be handled by ignoring the whole chunk as though it were an unknown chunk type. (This recommendation assumes that the chunk's CRC has been verified. In decoders that do not check CRCs, it is safer to treat any unexpected value as indicating a corrupted file.)

10.2. Pixel dimensions

Non-square pixels can be represented (see the pHYs chunk), but viewers are not required to account for them; a viewer can present any PNG file as though its pixels are square.

Conversely, viewers running on display hardware with non-square pixels are strongly encouraged to rescale images for proper display.

10.3. Truecolor image handling

To achieve PNG's goal of universal interchangeability, decoders are required to accept all types of PNG image: indexed-color, truecolor, and grayscale. Viewers running on indexed-color display hardware need to be able to reduce truecolor images to indexed format for viewing. This process is usually called "color quantization".

A simple, fast way of doing this is to reduce the image to a fixed palette. Palettes with uniform color spacing ("color cubes") are usually used to minimize the per-pixel computation. For photograph-like images, dithering is recommended to avoid ugly contours in what should be smooth gradients; however, dithering introduces graininess that can be objectionable.

The quality of rendering can be improved substantially by using a palette chosen specifically for the image, since a color cube usually has numerous entries that are unused in any particular image. This approach requires more work, first in choosing the palette, and second in mapping individual pixels to the closest available color. PNG allows the encoder to supply a suggested palette in a PLTE chunk, but not all encoders will do so, and the suggested palette may be unsuitable in any case (it may have too many or too few colors). High-quality viewers will therefore need to have a palette selection routine at hand. A large lookup table is usually the most feasible way of mapping individual pixels to palette entries with adequate speed.

Numerous implementations of color quantization are available. The PNG reference implementation, libpng, includes code for the purpose.

10.4. Bit depth rescaling

Decoders may wish to scale PNG data to a lesser bit depth (sample precision) for display. For example, 16-bit data will need to be reduced to 8-bit depth for use on most present-day display hardware. Reduction of 8-bit data to 5-bit depth is also common.

The most accurate scaling is achieved by the linear equation

```
output = ROUND(input * MAXOUTSAMPLE / MAXINSAMPLE)
```

where

```
MAXINSAMPLE = (2^bitdepth)-1
MAXOUTSAMPLE = (2^desired_bitdepth)-1
```

A slightly less accurate conversion is achieved by simply shifting right by bitdepth-desired_bitdepth places. For example, to reduce 16-bit samples to 8-bit, one need only discard the low-order byte. In many situations the shift method is sufficiently accurate for display purposes, and it is certainly much faster. (But if gamma correction is being done, sample rescaling can be merged into the gamma correction lookup table, as is illustrated in Decoder gamma handling, Section 10.5.)

When an sBIT chunk is present, the original pre-PNG data can be recovered by shifting right to the bit depth specified by sBIT. Note that linear scaling will not necessarily reproduce the original data, because the encoder is not required to have used linear scaling to scale the data up. However, the encoder is required to have used a method that preserves the high-order bits, so shifting always works. This is the only case in which shifting might be said to be more accurate than linear scaling.

When comparing pixel values to tRNS chunk values to detect

transparent pixels, it is necessary to do the comparison exactly. Therefore, transparent pixel detection must be done before reducing sample precision.

10.5. Decoder gamma handling

See Gamma Tutorial (Chapter 13) if you aren't already familiar with gamma issues.

To produce correct tone reproduction, a good image display program must take into account the gammas of the image file and the display device, as well as the viewing_gamma appropriate to the lighting conditions near the display. This can be done by calculating

```
gbright = sampleval / MAXSAMPLEVAL
bright = gbright ^ (1.0 / file_gamma)
vbright = bright ^ viewing_gamma
gcvideo = vbright ^ (1.0 / display_gamma)
fbval = ROUND(gcvideo * MAXFBVAL)
```

where MAXSAMPLEVAL is the maximum sample value in the file (255 for 8-bit, 65535 for 16-bit, etc), MAXFBVAL is the maximum value of a frame buffer sample (255 for 8-bit, 31 for 5-bit, etc), sampleval is the value of the sample in the PNG file, and fbval is the value to write into the frame buffer. The first line converts from integer samples into a normalized 0 to 1 floating point value, the second undoes the gamma encoding of the image file to produce a linear intensity value, the third adjusts for the viewing conditions, the fourth corrects for the display system's gamma value, and the fifth converts to an integer frame buffer sample. In practice, the second through fourth lines can be merged into

```
gcvideo = gbright^(viewing_gamma / (file_gamma*display_gamma))
```

so as to perform only one power calculation. For color images, the

entire calculation is performed separately for R, G, and B values.

It is not necessary to perform transcendental math for every pixel. Instead, compute a lookup table that gives the correct output value for every possible sample value. This requires only 256 calculations per image (for 8-bit accuracy), not one or three calculations per pixel. For an indexed-color image, a one-time correction of the palette is sufficient, unless the image uses transparency and is being displayed against a nonuniform background.

In some cases even the cost of computing a gamma lookup table may be a concern. In these cases, viewers are encouraged to have precomputed gamma correction tables for file_gamma values of 1.0 and 0.5 with some reasonable choice of viewing_gamma and

display_gamma, and to use the table closest to the gamma indicated in the file. This will produce acceptable results for the majority of real files.

When the incoming image has unknown gamma (no gAMA chunk), choose a likely default file_gamma value, but allow the user to select a new one if the result proves too dark or too light.

In practice, it is often difficult to determine what value of display_gamma should be used. In systems with no built-in gamma correction, the display_gamma is determined entirely by the CRT. Assuming a CRT_gamma of 2.5 is recommended, unless you have detailed calibration measurements of this particular CRT available.

However, many modern frame buffers have lookup tables that are used to perform gamma correction, and on these systems the display_gamma value should be the gamma of the lookup table and CRT combined. You may not be able to find out what the lookup table contains from within an image viewer application, so you may have to ask the user what the system's gamma value is. Unfortunately, different manufacturers use different ways of specifying what should go into the lookup table, so interpretation of the system gamma value is system-dependent. Gamma Tutorial (Chapter 13) gives some examples.

The response of real displays is actually more complex than can be described by a single number (display_gamma). If actual measurements of the monitor's light output as a function of voltage input are available, the fourth and fifth lines of the computation above can be replaced by a lookup in these measurements, to find the actual frame buffer value that most nearly gives the desired brightness.

The value of viewing_gamma depends on lighting conditions; see Gamma Tutorial (Chapter 13) for more detail. Ideally, a viewer would allow the user to specify viewing_gamma, either directly numerically, or via selecting from "bright surround", "dim surround", and "dark surround" conditions. Viewers that don't want to do this should just assume a value for viewing_gamma of 1.0, since most computer displays live in brightly-lit rooms.

When viewing images that are digitized from video, or that are destined to become video frames, the user might want to set the viewing_gamma to about 1.25 regardless of the actual level of room lighting. This value of viewing_gamma is "built into" NTSC video practice, and displaying an image with that viewing_gamma allows the user to see what a TV set would show under the current room lighting conditions. (This is not the same thing as trying to obtain the most accurate rendition of the content of the scene,

which would require adjusting viewing_gamma to correspond to the room lighting level.) This is another reason viewers might want

to allow users to adjust viewing_gamma directly.

10.6. Decoder color handling

See Color Tutorial (Chapter 14) if you aren't already familiar with color issues.

In many cases, decoders will treat image data in PNG files as device-dependent RGB data and display it without modification (except for appropriate gamma correction). This provides the fastest display of PNG images. But unless the viewer uses exactly the same display hardware as the original image author used, the colors will not be exactly the same as the original author saw, particularly for darker or near-neutral colors. The cHRM chunk provides information that allows closer color matching than that provided by gamma correction alone.

Decoders can use the cHRM data to transform the image data from RGB to XYZ and thence into a perceptually linear color space such as CIE LAB. They can then partition the colors to generate an optimal palette, because the geometric distance between two colors in CIE LAB is strongly related to how different those colors appear (unlike, for example, RGB or XYZ spaces). The resulting palette of colors, once transformed back into RGB color space, could be used for display or written into a PLTE chunk.

Decoders that are part of image processing applications might also transform image data into CIE LAB space for analysis.

In applications where color fidelity is critical, such as product design, scientific visualization, medicine, architecture, or advertising, decoders can transform the image data from source_RGB to the display_RGB space of the monitor used to view the image. This involves calculating the matrix to go from source_RGB to XYZ and the matrix to go from XYZ to display_RGB, then combining them to produce the overall transformation. The decoder is responsible for implementing gamut mapping.

Decoders running on platforms that have a Color Management System (CMS) can pass the image data, gAMA and cHRM values to the CMS for display or further processing.

Decoders that provide color printing facilities can use the facilities in Level 2 PostScript to specify image data in calibrated RGB space or in a device-independent color space such as XYZ. This will provide better color fidelity than a simple RGB to CMYK conversion. The PostScript Language Reference manual gives examples of this process [POSTSCRIPT]. Such decoders are responsible for implementing gamut mapping between source_RGB (specified in the cHRM chunk) and the target printer. The PostScript interpreter is then responsible for producing the required colors.

Decoders can use the cHRM data to calculate an accurate grayscale representation of a color image. Conversion from RGB to gray is simply a case of calculating the Y (luminance) component of XYZ,

which is a weighted sum of the R G and B values. The weights depend on the monitor type, i.e., the values in the cHRM chunk. Decoders may wish to do this for PNG files with no cHRM chunk. In that case, a reasonable default would be the CCIR 709 primaries [ITU-BT709]. Do not use the original NTSC primaries, unless you really do have an image color-balanced for such a monitor. Few monitors ever used the NTSC primaries, so such images are probably nonexistent these days.

10.7. Background color

The background color given by bKGD will typically be used to fill unused screen space around the image, as well as any transparent pixels within the image. (Thus, bKGD is valid and useful even when the image does not use transparency.) If no bKGD chunk is present, the viewer must make its own decision about a suitable background color.

Viewers that have a specific background against which to present the image (such as Web browsers) will ignore the bKGD chunk, in effect overriding bKGD with their preferred background color or background image.

The background color given by bKGD is not to be considered transparent, even if it happens to match the color given by tRNS (or, in the case of an indexed-color image, refers to a palette index that is marked as transparent by tRNS). Otherwise one would have to imagine something "behind the background" to composite against. The background color is either used as background or ignored; it is not an intermediate layer between the PNG image and some other background.

Indeed, it will be common that bKGD and tRNS specify the same color, since then a decoder that does not implement transparency processing will give the intended display, at least when no partially-transparent pixels are present.

10.8. Alpha channel processing

In the most general case, the alpha channel can be used to composite a foreground image against a background image; the PNG file defines the foreground image and the transparency mask, but not the background image. Decoders are not required to support this most general case. It is expected that most will be able to support compositing against a single background color, however.

The equation for computing a composited sample value is

```
output = alpha * foreground + (1-alpha) * background
```


where alpha and the input and output sample values are expressed as fractions in the range 0 to 1. This computation should be performed with linear (non-gamma-encoded) sample values. For color images, the computation is done separately for R, G, and B samples.

The following code illustrates the general case of compositing a foreground image over a background image. It assumes that you have the original pixel data available for the background image, and that output is to a frame buffer for display. Other variants are possible; see the comments below the code. The code allows the bit depths and gamma values of foreground image, background image, and frame buffer/CRT all to be different. Don't assume they are the same without checking.

This code is standard C, with line numbers added for reference in

the comments below.

```
    01  int foreground[4];  /* image pixel: R, G, B, A */
    02  int background[3];  /* background pixel: R, G, B */
    03  int fbpix[3];       /* frame buffer pixel */
    04  int fg_maxsample;   /* foreground max sample */
    05  int bg_maxsample;   /* background max sample */
    06  int fb_maxsample;   /* frame buffer max sample */
    07  int ialpha;
    08  float alpha, compalpha;
    09  float gamfg, linfg, gambg, linbg, comppix, gcvideo;

        /* Get max sample values in data and frame buffer */
    10  fg_maxsample = (1 << fg_bit_depth) - 1;
    11  bg_maxsample = (1 << bg_bit_depth) - 1;
    12  fb_maxsample = (1 << frame_buffer_bit_depth) - 1;
        /*
         * Get integer version of alpha.
         * Check for opaque and transparent special cases;
         * no compositing needed if so.
         *
         * We show the whole gamma decode/correct process in
         * floating point, but it would more likely be done
         * with lookup tables.
         */
    13  ialpha = foreground[3];
```



```
    14  if (ialpha == 0) {
            /*
             * Foreground image is transparent here.
             * If the background image is already in the frame
             * buffer, there is nothing to do.
             */
    15      ;
    16  } else if (ialpha == fg_maxsample) {
            /*
             * Copy foreground pixel to frame buffer.
             */
    17      for (i = 0; i < 3; i++) {
    18          gamfg = (float) foreground[i] / fg_maxsample;
    19          linfg = pow(gamfg, 1.0/fg_gamma);
    20          comppix = linfg;
    21          gcvideo = pow(comppix,viewing_gamma/display_gamma);
    22          fbpix[i] = (int) (gcvideo * fb_maxsample + 0.5);
    23      }
    24  } else {
            /*
             * Compositing is necessary.
             * Get floating-point alpha and its complement.
             * Note: alpha is always linear; gamma does not
             * affect it.
             */
    25      alpha = (float) ialpha / fg_maxsample;
    26      compalpha = 1.0 - alpha;

    27      for (i = 0; i < 3; i++) {
```

```
                /*
                 * Convert foreground and background to floating
                 * point, then linearize (undo gamma encoding).
                 */
   28           gamfg = (float) foreground[i] / fg_maxsample;
   29           linfg = pow(gamfg, 1.0/fg_gamma);
   30           gambg = (float) background[i] / bg_maxsample;
   31           linbg = pow(gambg, 1.0/bg_gamma);
                /*
                 * Composite.
                 */
   32           comppix = linfg * alpha + linbg * compalpha;
                /*
                 * Gamma correct for display.
                 * Convert to integer frame buffer pixel.
                 */
   33           gcvideo = pow(comppix,viewing_gamma/display_gamma);
   34           fbpix[i] = (int) (gcvideo * fb_maxsample + 0.5);
   35       }
   36   }
```

Variations:

* If output is to another PNG image file instead of a frame buffer, lines 21, 22, 33, and 34 should be changed to be something like

```
   /*
    * Gamma encode for storage in output file.
    * Convert to integer sample value.
    */
   gamout = pow(comppix, outfile_gamma);
   outpix[i] = (int) (gamout * out_maxsample + 0.5);
```

  Also, it becomes necessary to process background pixels when alpha is zero, rather than just skipping pixels. Thus, line 15 must be replaced by copies of lines 17-23, but processing background instead of foreground pixel values.
* If the bit depth of the output file, foreground file, and background file are all the same, and the three gamma values also match, then the no-compositing code in lines 14-23 reduces to nothing more than copying pixel values from the input file to the output file if alpha is one, or copying pixel values from background to output file if alpha is zero. Since alpha is typically either zero or one for the vast majority of pixels in an image, this is a great savings. No gamma computations are needed for most pixels.
* When the bit depths and gamma values all match, it may appear attractive to skip the gamma decoding and encoding (lines 28-31, 33-34) and just perform line 32 using gamma-encoded sample values. Although this doesn't hurt image quality too badly, the time savings are small if alpha values of zero and one are special-cased as recommended here.
* If the original pixel values of the background image are no longer available, only processed frame buffer pixels left by display of the background image, then lines 30 and 31 must extract intensity from the frame buffer pixel values using code like

```
   /*
    * Decode frame buffer value back into linear space.
    */
   gcvideo = (float) fbpix[i] / fb_maxsample;
   linbg = pow(gcvideo, display_gamma / viewing_gamma);
```

However, some roundoff error can result, so it is better to have the original background pixels available if at all possible.
* Note that lines 18-22 are performing exactly the same gamma computation that is done when no alpha channel is present. So, if you handle the no-alpha case with a lookup table, you can use the same lookup table here. Lines 28-31 and 33-34 can also be done with (different) lookup tables.
* Of course, everything here can be done in integer

arithmetic. Just be careful to maintain sufficient precision all the way through.

Note: in floating point, no overflow or underflow checks are needed, because the input sample values are guaranteed to be between 0 and 1, and compositing always yields a result that is in between the input values (inclusive). With integer arithmetic, some roundoff-error analysis might be needed to guarantee no overflow or underflow.

When displaying a PNG image with full alpha channel, it is important to be able to composite the image against some background, even if it's only black. Ignoring the alpha channel will cause PNG images that have been converted from an associated-alpha representation to look wrong. (Of course, if the alpha channel is a separate transparency mask, then ignoring alpha is a useful option: it allows the hidden parts of the image to be recovered.)

Even if the decoder author does not wish to implement true compositing logic, it is simple to deal with images that contain only zero and one alpha values. (This is implicitly true for grayscale and truecolor PNG files that use a tRNS chunk; for indexed-color PNG files, it is easy to check whether tRNS contains any values other than 0 and 255.) In this simple case, transparent pixels are replaced by the background color, while others are unchanged. If a decoder contains only this much transparency capability, it should deal with a full alpha channel by treating all nonzero alpha values as fully opaque; that is, do not replace partially transparent pixels by the background. This approach will not yield very good results for images converted from associated-alpha formats, but it's better than doing nothing.

10.9. Progressive display

When receiving images over slow transmission links, decoders can improve perceived performance by displaying interlaced images progressively. This means that as each pass is received, an approximation to the complete image is displayed based on the data received so far. One simple yet pleasing effect can be obtained by expanding each received pixel to fill a rectangle covering the yet-to-be-transmitted pixel positions below and to the right of the received pixel. This process can be described by the following pseudocode:

```
Starting_Row [1..7] =  { 0, 0, 4, 0, 2, 0, 1 }
Starting_Col [1..7] =  { 0, 4, 0, 2, 0, 1, 0 }
Row_Increment [1..7] = { 8, 8, 8, 4, 4, 2, 2 }
Col_Increment [1..7] = { 8, 8, 4, 4, 2, 2, 1 }
Block_Height [1..7] =  { 8, 8, 4, 4, 2, 2, 1 }
Block_Width [1..7] =   { 8, 4, 4, 2, 2, 1, 1 }
```

```
pass := 1
while pass <= 7
begin
    row := Starting_Row[pass]

    while row < height
    begin
        col := Starting_Col[pass]

        while col < width
        begin
            visit (row, col,
                   min (Block_Height[pass], height - row),
                   min (Block_Width[pass], width - col))
            col := col + Col_Increment[pass]
        end
        row := row + Row_Increment[pass]
    end

    pass := pass + 1
end
```

Here, the function "visit(row,column,height,width)" obtains the next transmitted pixel and paints a rectangle of the specified height and width, whose upper-left corner is at the specified row and column, using the color indicated by the pixel. Note that row and column are measured from 0,0 at the upper left corner.

If the decoder is merging the received image with a background image, it may be more convenient just to paint the received pixel positions; that is, the "visit()" function sets only the pixel at the specified row and column, not the whole rectangle. This produces a "fade-in" effect as the new image gradually replaces the old. An advantage of this approach is that proper alpha or transparency processing can be done as each pixel is replaced. Painting a rectangle as described above will overwrite background-image pixels that may be needed later, if the pixels eventually received for those positions turn out to be wholly or partially transparent. Of course, this is only a problem if the background image is not stored anywhere offscreen.

10.10. Suggested-palette and histogram usage

In truecolor PNG files, the encoder may have provided a suggested PLTE chunk for use by viewers running on indexed-color hardware.

If the image has a tRNS chunk, the viewer will need to adapt the suggested palette for use with its desired background color. To do this, replace the palette entry closest to the tRNS color with the desired background color; or just add a palette entry for the background color, if the viewer can handle more colors than there are PLTE entries.

For images of color type 6 (truecolor with alpha channel), any suggested palette should have been designed for display of the image against a uniform background of the color specified by bKGD. Viewers should probably ignore the palette if they intend to use a different background, or if the bKGD chunk is missing. Viewers can use a suggested palette for display against a different background than it was intended for, but the results may not be very good.

If the viewer presents a transparent truecolor image against a background that is more complex than a single color, it is unlikely that the suggested palette will be optimal for the composite image. In this case it is best to perform a truecolor compositing step on the truecolor PNG image and background image, then color-quantize the resulting image.

The histogram chunk is useful when the viewer cannot provide as many colors as are used in the image's palette. If the viewer is only short a few colors, it is usually adequate to drop the least-used colors from the palette. To reduce the number of colors substantially, it's best to choose entirely new representative colors, rather than trying to use a subset of the existing palette. This amounts to performing a new color quantization step; however, the existing palette and histogram can be used as the input data, thus avoiding a scan of the image data.

If no palette or histogram chunk is provided, a decoder can develop its own, at the cost of an extra pass over the image data. Alternatively, a default palette (probably a color cube) can be used.

See also Recommendations for Encoders: Suggested palettes (Section 9.5).

10.11. Text chunk processing

If practical, decoders should have a way to display to the user all tEXt and zTXt chunks found in the file. Even if the decoder does not recognize a particular text keyword, the user might be able to understand it.

PNG text is not supposed to contain any characters outside the ISO 8859-1 "Latin-1" character set (that is, no codes 0-31 or 127-159), except for the newline character (decimal 10). But decoders might encounter such characters anyway. Some of these characters can be safely displayed (e.g., TAB, FF, and CR, decimal 9, 12, and 13, respectively), but others, especially the ESC character (decimal 27), could pose a security hazard because unexpected actions may be taken by display hardware or software. To prevent such hazards, decoders should not attempt to directly display any non-Latin-1 characters (except for newline and perhaps TAB, FF, CR) encountered in a tEXt or zTXt chunk. Instead, ignore them or

display them in a visible notation such as "\nnn". See Security considerations (Section 8.5).

Even though encoders are supposed to represent newlines as LF, it is recommended that decoders not rely on this; it's best to recognize all the common newline combinations (CR, LF, and CR-LF) and display each as a single newline. TAB can be expanded to the proper number of spaces needed to arrive at a column multiple of 8.

Decoders running on systems with non-Latin-1 character set encoding should provide character code remapping so that Latin-1 characters are displayed correctly. Some systems may not provide all the characters defined in Latin-1. Mapping unavailable characters to a visible notation such as "\nnn" is a good fallback. In particular, character codes 127-255 should be displayed only if they are printable characters on the decoding system. Some systems may interpret such codes as control characters; for security, decoders running on such systems should not display such characters literally.

Decoders should be prepared to display text chunks that contain any number of printing characters between newline characters, even though encoders are encouraged to avoid creating lines in excess of 79 characters.

11. Glossary

Alpha
: A value representing a pixel's degree of transparency. The more transparent a pixel, the less it hides the background against which the image is presented. In PNG, alpha is really the degree of opacity: zero alpha represents a completely transparent pixel, maximum alpha represents a completely opaque pixel. But most people refer to alpha as providing transparency information, not opacity information, and we continue that custom here.

Ancillary chunk
: A chunk that provides additional information. A decoder can still produce a meaningful image, though not necessarily the best possible image, without processing the chunk.

Byte
: Eight bits; also called an octet.

Channel
: The set of all samples of the same kind within an image; for example, all the blue samples in a truecolor image. (The term "component" is also used, but not in this specification.) A sample is the intersection of a channel and a pixel.

Chunk
: A section of a PNG file. Each chunk has a type indicated by its chunk type name. Most types of chunks also include some data. The format and meaning of the data within the chunk are determined by the type name.

Chromaticity
: A pair of values x,y that precisely specify the hue, though not the absolute brightness, of a perceived color.

Composite
: As a verb, to form an image by merging a foreground image and a background image, using transparency information to determine where the background should be visible. The foreground image is said to be "composited against" the background.

CRC
: Cyclic Redundancy Check. A CRC is a type of check value designed to catch most transmission errors. A decoder calculates the CRC for the received data and compares it to the CRC that the encoder calculated, which is appended to the data. A mismatch indicates that the data was corrupted in transit.

CRT
: Cathode Ray Tube: a common type of computer display hardware.

Critical chunk
: A chunk that must be understood and processed by the decoder in order to produce a meaningful image from a PNG file.

Datastream
: A sequence of bytes. This term is used rather than "file" to describe a byte sequence that is only a portion of a file. We also use it to emphasize that a PNG image might be generated and

consumed "on the fly", never appearing in a stored file at all.

Deflate
The name of the compression algorithm used in standard PNG files, as well as in zip, gzip, pkzip, and other compression programs. Deflate is a member of the LZ77 family of compression methods.

Filter
A transformation applied to image data in hopes of improving its compressibility. PNG uses only lossless (reversible) filter algorithms.

Frame buffer
The final digital storage area for the image shown by a computer display. Software causes an image to appear onscreen by loading it into the frame buffer.

Gamma
The brightness of mid-level tones in an image. More precisely, a parameter that describes the shape of the transfer function for one or more stages in an imaging pipeline. The transfer function is given by the expression

```
output = input ^ gamma
```

where both input and output are scaled to the range 0 to 1.

Grayscale
An image representation in which each pixel is represented by a single sample value representing overall luminance (on a scale from black to white). PNG also permits an alpha sample to be stored for each pixel of a grayscale image.

Indexed color
An image representation in which each pixel is represented by a single sample that is an index into a palette or lookup table. The selected palette entry defines the actual color of the pixel.

Lossless compression
Any method of data compression that guarantees the original data can be reconstructed exactly, bit-for-bit.

Lossy compression
Any method of data compression that reconstructs the original data approximately, rather than exactly.

LSB
Least Significant Byte of a multi-byte value.

Luminance
Perceived brightness, or grayscale level, of a color. Luminance and chromaticity together fully define a perceived color.

LUT
Look Up Table. In general, a table used to transform data. In frame buffer hardware, a LUT can be used to map indexed-color pixels into a selected set of truecolor values, or to perform gamma correction. In software, a LUT can be used as a fast way of implementing any one-variable mathematical function.

MSB
Most Significant Byte of a multi-byte value.

Palette

The set of colors available in an indexed-color image. In PNG, a palette is an array of colors defined by red, green, and blue samples. (Alpha values can also be defined for palette entries, via the tRNS chunk.)

Pixel
The information stored for a single grid point in the image. The complete image is a rectangular array of pixels.

PNG editor
A program that modifies a PNG file and preserves ancillary information, including chunks that it does not recognize. Such a program must obey the rules given in Chunk Ordering Rules (Chapter 7).

Sample
A single number in the image data; for example, the red value of a pixel. A pixel is composed of one or more samples. We use "sample" both for color values and for the palette index values of an indexed-color image.

Scanline
One horizontal row of pixels within an image.

Truecolor
An image representation in which pixel colors are defined by storing three samples for each pixel, representing red, green, and blue intensities respectively. PNG also permits an alpha sample to be stored for each pixel of a truecolor image.

White point
The chromaticity of a computer display's nominal white value.

zlib
A particular format for data that has been compressed using deflate-style compression. Also the name of a library implementing this method. PNG implementations need not use the zlib library, but they must conform to its format for compressed data.

x^y
Exponentiation; x raised to the power y. C programmers should be careful not to misread this notation as exclusive-or. Note that in gamma-related calculations, zero raised to any power is valid and should give a zero result.

12. Appendix: Rationale

(This appendix is not part of the formal PNG specification.)

This appendix gives the reasoning behind some of the design decisions in PNG. Many of these decisions were the subject of considerable debate. The authors freely admit that another group might have made different decisions; however, we believe that our choices are defensible and consistent.

12.1. Why a new file format?

Does the world really need yet another graphics format? We believe so. GIF is no longer freely usable, but no other commonly used format can directly replace it, as is discussed in more detail below. We might have used an adaptation of an existing format, for example GIF with an unpatented compression scheme. But this would require new code anyway; it would not be all that much easier to implement than a whole new file format. (PNG is designed to be simple to implement, with the exception of the compression engine, which would be needed in any case.) We feel that this is an excellent opportunity to design a new format that fixes some of the known limitations of GIF.

12.2. Why these features?

The features chosen for PNG are intended to address the needs of applications that previously used the special strengths of GIF. In particular, GIF is well adapted for online communications because of its streamability and progressive display capability. PNG shares those attributes.

We have also addressed some of the widely known shortcomings of GIF. In particular, PNG supports truecolor images. We know of no widely used image format that losslessly compresses truecolor images as effectively as PNG does. We hope that PNG will make use of truecolor images more practical and widespread.

Some form of transparency control is desirable for applications in which images are displayed against a background or together with other images. GIF provided a simple transparent-color specification for this purpose. PNG supports a full alpha channel as well as transparent-color specifications. This allows both highly flexible transparency and compression efficiency.

Robustness against transmission errors has been an important consideration. For example, images transferred across Internet are often mistakenly processed as text, leading to file corruption. PNG is designed so that such errors can be detected quickly and reliably.

PNG has been expressly designed not to be completely dependent on

a single compression technique. Although deflate/inflate compression is mentioned in this document, PNG would still exist without it.

12.3. Why not these features?

Some features have been deliberately omitted from PNG. These choices were made to simplify implementation of PNG, promote portability and interchangeability, and make the format as simple and foolproof as possible for users. In particular:

* There is no uncompressed variant of PNG. It is possible to store uncompressed data by using only uncompressed deflate

blocks (a feature normally used to guarantee that deflate does not make incompressible data much larger). However, any software that does not support full deflate/inflate will not be considered compliant with the PNG standard. The two most important features of PNG---portability and compression---are absolute requirements for online applications, and users demand them. Failure to support full deflate/inflate compromises both of these objectives.

* There is no lossy compression in PNG. Existing formats such as JFIF already handle lossy compression well. Furthermore, available lossy compression methods (e.g., JPEG) are far from foolproof --- a poor choice of quality level can ruin an image. To avoid user confusion and unintentional loss of information, we feel it is best to keep lossy and lossless formats strictly separate. Also, lossy compression is complex to implement. Adding JPEG support to a PNG decoder might increase its size by an order of magnitude. This would certainly cause some decoders to omit support for the feature, which would destroy our goal of interchangeability.
* There is no support for CMYK or other unusual color spaces. Again, this is in the name of promoting portability. CMYK, in particular, is far too device-dependent to be useful as a portable image representation.
* There is no standard chunk for thumbnail views of images. In discussions with software vendors who use thumbnails in their products, it has become clear that most would not use a "standard" thumbnail chunk. For one thing, every vendor has a different idea of what the dimensions and characteristics of a thumbnail should be. Also, some vendors keep thumbnails in separate files to accommodate varied image formats; they are not going to stop doing that simply because of a thumbnail chunk in one new format. Proprietary chunks containing vendor-specific thumbnails appear to be more practical than a common thumbnail format.

It is worth noting that private extensions to PNG could easily add these features. We will not, however, include them as part of the basic PNG standard.

Basic PNG also does not support multiple images in one file. This restriction is a reflection of the reality that many applications do not need and will not support multiple images per file. (While the GIF standard nominally allows multiple images per file, few applications actually support it.) In any case, single images are a fundamentally different sort of object from sequences of images. Rather than make false promises of interchangeability, we have drawn a clear distinction between single-image and multi-image formats. PNG is a single-image format.

12.4. Why not use format X?

Numerous existing formats were considered before deciding to develop PNG. None could meet the requirements we felt were important for PNG.

GIF is no longer suitable as a universal standard because of legal entanglements. Although just replacing GIF's compression method would avoid that problem, GIF does not support truecolor images, alpha channels, or gamma correction. The spec has more subtle problems too. Only a small subset of the GIF89 spec is actually portable across a variety of implementations, but there is no codification of the most portable part of the spec.

TIFF is far too complex to meet our goals of simplicity and interchangeability. Defining a TIFF subset would meet that

objection, but would frustrate users making the reasonable assumption that a file saved as TIFF from their existing software would load into a program supporting our flavor of TIFF. Furthermore, TIFF is not designed for stream processing, has no provision for progressive display, and does not currently provide any good, legally unencumbered, lossless compression method.

IFF has also been suggested, but is not suitable in detail: available image representations are too machine-specific or not adequately compressed. The overall chunk structure of IFF is a useful concept that PNG has liberally borrowed from, but we did not attempt to be bit-for-bit compatible with IFF chunk structure. Again this is due to detailed issues, notably the fact that IFF FORMs are not designed to be serially writable.

Lossless JPEG is not suitable because it does not provide for the storage of indexed-color images. Furthermore, its lossless truecolor compression is often inferior to that of PNG.

12.5. Byte order

It has been asked why PNG uses network byte order. We have selected one byte ordering and used it consistently. Which order in particular is of little relevance, but network byte order has the advantage that routines to convert to and from it are already available on any platform that supports TCP/IP networking,

including all PC platforms. The functions are trivial and will be included in the reference implementation.

12.6. Interlacing

PNG's two-dimensional interlacing scheme is more complex to implement than GIF's line-wise interlacing. It also costs a little more in file size. However, it yields an initial image eight times faster than GIF (the first pass transmits only 1/64th of the pixels, compared to 1/8th for GIF). Although this initial image is coarse, it is useful in many situations. For example, if the image is a World Wide Web imagemap that the user has seen before, PNG's first pass is often enough to determine where to click. The PNG scheme also looks better than GIF's, because horizontal and vertical resolution never differ by more than a factor of two; this avoids the odd "stretched" look seen when interlaced GIFs are filled in by replicating scanlines. Preliminary results show that small text in an interlaced PNG image is typically readable about twice as fast as in an equivalent GIF, i.e., after PNG's fifth pass or 25% of the image data, instead of after GIF's third pass or 50%. This is again due to PNG's more balanced increase in resolution.

12.7. Why gamma?

It might seem natural to standardize on storing sample values that are linearly proportional to light intensity (that is, have gamma of 1.0). But in fact, it is common for images to have a gamma of less than 1. There are three good reasons for this:

* For reasons detailed in Gamma Tutorial (Chapter 13), all video cameras apply a "gamma correction" function to the intensity information. This causes the video signal to have a gamma of about 0.5 relative to the light intensity in the original scene. Thus, images obtained by frame-grabbing video already have a gamma of about 0.5.
* The human eye has a nonlinear response to intensity, so linear encoding of samples either wastes sample codes in bright areas of the image, or provides too few sample codes

to avoid banding artifacts in dark areas of the image, or both. At least 12 bits per sample are needed to avoid visible artifacts in linear encoding with a 100:1 image intensity range. An image gamma in the range 0.3 to 0.5 allocates sample values in a way that roughly corresponds to the eye's response, so that 8 bits/sample are enough to avoid artifacts caused by insufficient sample precision in almost all images. This makes "gamma encoding" a much better way of storing digital images than the simpler linear encoding.

* Many images are created on PCs or workstations with no gamma correction hardware and no software willing to provide gamma correction either. In these cases, the images have had

their lighting and color chosen to look best on this platform --- they can be thought of as having "manual" gamma correction built in. To see what the image author intended, it is necessary to treat such images as having a file_gamma value in the range 0.4-0.6, depending on the room lighting level that the author was working in.

In practice, image gamma values around 1.0 and around 0.5 are both widely found. Older image standards such as GIF often do not account for this fact. The JFIF standard specifies that images in that format should use linear samples, but many JFIF images found on the Internet actually have a gamma somewhere near 0.4 or 0.5. The variety of images found and the variety of systems that people display them on have led to widespread problems with images appearing "too dark" or "too light".

PNG expects viewers to compensate for image gamma at the time that the image is displayed. Another possible approach is to expect encoders to convert all images to a uniform gamma at encoding time. While that method would speed viewers slightly, it has fundamental flaws:

* Gamma correction is inherently lossy due to quantization and roundoff error. Requiring conversion at encoding time thus causes irreversible loss. Since PNG is intended to be a lossless storage format, this is undesirable; we should store unmodified source data.
* The encoder might not know the source gamma value. If the decoder does gamma correction at viewing time, it can adjust the gamma (change the displayed brightness) in response to feedback from a human user. The encoder has no such recourse.
* Whatever "standard" gamma we settled on would be wrong for some displays. Hence viewers would still need gamma correction capability.

Since there will always be images with no gamma or an incorrect recorded gamma, good viewers will need to incorporate gamma adjustment code anyway. Gamma correction at viewing time is thus the right way to go.

See Gamma Tutorial (Chapter 13) for more information.

12.8. Non-premultiplied alpha

PNG uses "unassociated" or "non-premultiplied" alpha so that images with separate transparency masks can be stored losslessly. Another common technique, "premultiplied alpha", stores pixel values premultiplied by the alpha fraction; in effect, the image is already composited against a black background. Any image data hidden by the transparency mask is irretrievably lost by that method, since multiplying by a zero alpha value always produces

zero.

Some image rendering techniques generate images with premultiplied alpha (the alpha value actually represents how much of the pixel is covered by the image). This representation can be converted to PNG by dividing the sample values by alpha, except where alpha is zero. The result will look good if displayed by a viewer that handles alpha properly, but will not look very good if the viewer ignores the alpha channel.

Although each form of alpha storage has its advantages, we did not want to require all PNG viewers to handle both forms. We standardized on non-premultiplied alpha as being the lossless and more general case.

12.9. Filtering

PNG includes filtering capability because filtering can significantly reduce the compressed size of truecolor and grayscale images. Filtering is also sometimes of value on indexed-color images, although this is less common.

The filter algorithms are defined to operate on bytes, rather than pixels; this gains simplicity and speed with very little cost in compression performance. Tests have shown that filtering is usually ineffective for images with fewer than 8 bits per sample, so providing pixelwise filtering for such images would be pointless. For 16 bit/sample data, bytewise filtering is nearly as effective as pixelwise filtering, because MSBs are predicted from adjacent MSBs, and LSBs are predicted from adjacent LSBs.

The encoder is allowed to change filters for each new scanline. This creates no additional complexity for decoders, since a decoder is required to contain defiltering logic for every filter type anyway. The only cost is an extra byte per scanline in the pre-compression datastream. Our tests showed that when the same filter is selected for all scanlines, this extra byte compresses away to almost nothing, so there is little storage cost compared to a fixed filter specified for the whole image. And the potential benefits of adaptive filtering are too great to ignore. Even with the simplistic filter-choice heuristics so far discovered, adaptive filtering usually outperforms fixed filters. In particular, an adaptive filter can change behavior for successive passes of an interlaced image; a fixed filter cannot.

12.10. Text strings

Most graphics file formats include the ability to store some textual information along with the image. But many applications need more than that: they want to be able to store several identifiable pieces of text. For example, a database using PNG files to store medical X-rays would likely want to include

patient's name, doctor's name, etc. A simple way to do this in PNG would be to invent new private chunks holding text. The disadvantage of such an approach is that other applications would have no idea what was in those chunks, and would simply ignore them. Instead, we recommend that textual information be stored in

standard tEXt chunks with suitable keywords. Use of tEXt tells any PNG viewer that the chunk contains text that might be of interest to a human user. Thus, a person looking at the file with another viewer will still be able to see the text, and even understand what it is if the keywords are reasonably self-explanatory. (To this end, we recommend spelled-out keywords, not abbreviations that will be hard for a person to understand. Saving a few bytes on a keyword is false economy.)

The ISO 8859-1 (Latin-1) character set was chosen as a compromise between functionality and portability. Some platforms cannot display anything more than 7-bit ASCII characters, while others can handle characters beyond the Latin-1 set. We felt that Latin-1 represents a widely useful and reasonably portable character set. Latin-1 is a direct subset of character sets commonly used on popular platforms such as Microsoft Windows and X Windows. It can also be handled on Macintosh systems with a simple remapping of characters.

There is presently no provision for text employing character sets other than Latin-1. We recognize that the need for other character sets will increase. However, PNG already requires that programmers implement a number of new and unfamiliar features, and text representation is not PNG's primary purpose. Since PNG provides for the creation and public registration of new ancillary chunks of general interest, we expect that text chunks for other character sets, such as Unicode, eventually will be registered and increase gradually in popularity.

12.11. PNG file signature

The first eight bytes of a PNG file always contain the following values:

```
   (decimal)              137  80  78  71  13  10  26  10
   (hexadecimal)           89  50  4e  47  0d  0a  1a  0a
   (ASCII C notation)    \211   P   N   G  \r  \n \032 \n
```

This signature both identifies the file as a PNG file and provides for immediate detection of common file-transfer problems. The first two bytes distinguish PNG files on systems that expect the first two bytes to identify the file type uniquely. The first byte is chosen as a non-ASCII value to reduce the probability that a text file may be misrecognized as a PNG file; also, it catches bad file transfers that clear bit 7. Bytes two through four name the format. The CR-LF sequence catches bad file transfers that alter newline sequences. The control-Z character stops file

display under MS-DOS. The final line feed checks for the inverse of the CR-LF translation problem.

A decoder may further verify that the next eight bytes contain an IHDR chunk header with the correct chunk length; this will catch bad transfers that drop or alter null (zero) bytes.

Note that there is no version number in the signature, nor indeed anywhere in the file. This is intentional: the chunk mechanism provides a better, more flexible way to handle format extensions, as explained in Chunk naming conventions (Section 12.13).

12.12. Chunk layout

The chunk design allows decoders to skip unrecognized or uninteresting chunks: it is simply necessary to skip the appropriate number of bytes, as determined from the length field.

Limiting chunk length to (2^31)-1 bytes avoids possible problems for implementations that cannot conveniently handle 4-byte unsigned values. In practice, chunks will usually be much shorter than that anyway.

A separate CRC is provided for each chunk in order to detect badly-transferred images as quickly as possible. In particular, critical data such as the image dimensions can be validated before being used.

The chunk length is excluded from the CRC so that the CRC can be calculated as the data is generated; this avoids a second pass over the data in cases where the chunk length is not known in advance. Excluding the length from the CRC does not create any extra risk of failing to discover file corruption, since if the length is wrong, the CRC check will fail: the CRC will be computed on the wrong set of bytes and then be tested against the wrong value from the file.

12.13. Chunk naming conventions

The chunk naming conventions allow safe, flexible extension of the PNG format. This mechanism is much better than a format version number, because it works on a feature-by-feature basis rather than being an overall indicator. Decoders can process newer files if and only if the files use no unknown critical features (as indicated by finding unknown critical chunks). Unknown ancillary chunks can be safely ignored. We decided against having an overall format version number because experience has shown that format version numbers hurt portability as much as they help. Version numbers tend to be set unnecessarily high, leading to older decoders rejecting files that they could have processed (this was a serious problem for several years after the GIF89 spec came out, for example). Furthermore, private extensions can be

made either critical or ancillary, and standard decoders will react appropriately; overall version numbers are no help for private extensions.

A hypothetical chunk for vector graphics would be a critical chunk, since if ignored, important parts of the intended image would be missing. A chunk carrying the Mandelbrot set coordinates for a fractal image would be ancillary, since other applications could display the image without understanding what the image represents. In general, a chunk type should be made critical only if it is impossible to display a reasonable representation of the intended image without interpreting that chunk.

The public/private property bit ensures that any newly defined public chunk type name cannot conflict with proprietary chunks that could be in use somewhere. However, this does not protect users of private chunk names from the possibility that someone else may use the same chunk name for a different purpose. It is a good idea to put additional identifying information at the start of the data for any private chunk type.

When a PNG file is modified, certain ancillary chunks may need to be changed to reflect changes in other chunks. For example, a histogram chunk needs to be changed if the image data changes. If the file editor does not recognize histogram chunks, copying them blindly to a new output file is incorrect; such chunks should be dropped. The safe/unsafe property bit allows ancillary chunks to be marked appropriately.

Not all possible modification scenarios are covered by the safe/unsafe semantics. In particular, chunks that are dependent

on the total file contents are not supported. (An example of such a chunk is an index of IDAT chunk locations within the file: adding a comment chunk would inadvertently break the index.) Definition of such chunks is discouraged. If absolutely necessary for a particular application, such chunks can be made critical chunks, with consequent loss of portability to other applications. In general, ancillary chunks can depend on critical chunks but not on other ancillary chunks. It is expected that mutually dependent information should be put into a single chunk.

In some situations it may be unavoidable to make one ancillary chunk dependent on another. Although the chunk property bits are insufficient to represent this case, a simple solution is available: in the dependent chunk, record the CRC of the chunk depended on. It can then be determined whether that chunk has been changed by some other program.

The same technique can be useful for other purposes. For example, if a program relies on the palette being in a particular order, it can store a private chunk containing the CRC of the PLTE chunk. If this value matches when the file is again read in, then it

provides high confidence that the palette has not been tampered with. Note that it is not necessary to mark the private chunk unsafe-to-copy when this technique is used; thus, such a private chunk can survive other editing of the file.

12.14. Palette histograms

A viewer may not be able to provide as many colors as are listed in the image's palette. (For example, some colors could be reserved by a window system.) To produce the best results in this situation, it is helpful to have information about the frequency with which each palette index actually appears, in order to choose the best palette for dithering or to drop the least-used colors. Since images are often created once and viewed many times, it makes sense to calculate this information in the encoder, although it is not mandatory for the encoder to provide it.

Other image formats have usually addressed this problem by specifying that the palette entries should appear in order of frequency of use. That is an inferior solution, because it doesn't give the viewer nearly as much information: the viewer can't determine how much damage will be done by dropping the last few colors. Nor does a sorted palette give enough information to choose a target palette for dithering, in the case that the viewer must reduce the number of colors substantially. A palette histogram provides the information needed to choose such a target palette without making a pass over the image data.

13. Appendix: Gamma Tutorial

(This appendix is not part of the formal PNG specification.)

It would be convenient for graphics programmers if all of the components of an imaging system were linear. The voltage coming from an electronic camera would be directly proportional to the intensity (power) of light in the scene, the light emitted by a CRT would be directly proportional to its input voltage, and so on. However, real-world devices do not behave in this way. All CRT displays, almost all photographic film, and many electronic cameras have nonlinear signal-to-light-intensity or intensity-to-signal characteristics.

Fortunately, all of these nonlinear devices have a transfer function that is approximated fairly well by a single type of mathematical

function: a power function. This power function has the general equation

output = input ^ gamma

where ^ denotes exponentiation, and "gamma" (often printed using the Greek letter gamma, thus the name) is simply the exponent of the power function.

By convention, "input" and "output" are both scaled to the range 0..1, with 0 representing black and 1 representing maximum white (or red, etc). Normalized in this way, the power function is completely described by a single number, the exponent "gamma".

So, given a particular device, we can measure its output as a function of its input, fit a power function to this measured transfer function, extract the exponent, and call it gamma. We often say "this device has a gamma of 2.5" as a shorthand for "this device has a power-law response with an exponent of 2.5". We can also talk about the gamma of a mathematical transform, or of a lookup table in a frame buffer, so long as the input and output of the thing are related by the power-law expression above.

How do gammas combine?

Real imaging systems will have several components, and more than one of these can be nonlinear. If all of the components have transfer characteristics that are power functions, then the transfer function of the entire system is also a power function. The exponent (gamma) of the whole system's transfer function is just the product of all of the individual exponents (gammas) of the separate stages in the system.

Also, stages that are linear pose no problem, since a power function with an exponent of 1.0 is really a linear function. So a linear transfer function is just a special case of a power function, with a gamma of 1.0.

Thus, as long as our imaging system contains only stages with linear and power-law transfer functions, we can meaningfully talk about the gamma of the entire system. This is indeed the case with most real imaging systems.

What should overall gamma be?

If the overall gamma of an imaging system is 1.0, its output is linearly proportional to its input. This means that the ratio between the intensities of any two areas in the reproduced image will be the same as it was in the original scene. It might seem that this should always be the goal of an imaging system: to accurately reproduce the tones of the original scene. Alas, that is not the case.

When the reproduced image is to be viewed in "bright surround" conditions, where other white objects nearby in the room have about the same brightness as white in the image, then an overall gamma of 1.0 does indeed give real-looking reproduction of a natural scene. Photographic prints viewed under room light and computer displays in bright room light are typical "bright surround" viewing conditions.

However, sometimes images are intended to be viewed in "dark surround" conditions, where the room is substantially black except for the image. This is typical of the way movies and slides (transparencies) are viewed by projection. Under these circumstances, an accurate reproduction of the original scene results in an image that human viewers judge as "flat" and lacking in contrast. It turns out that the projected image needs to have a gamma of about 1.5 relative to the original scene for viewers to judge it "natural". Thus, slide film is designed to have a gamma of about 1.5, not 1.0.

There is also an intermediate condition called "dim surround", where the rest of the room is still visible to the viewer, but is noticeably darker than the reproduced image itself. This is typical of television viewing, at least in the evening, as well as subdued-light computer work areas. In dim surround conditions, the reproduced image needs to have a gamma of about 1.25 relative to the original scene in order to look natural.

The requirement for boosted contrast (gamma) in dark surround conditions is due to the way the human visual system works, and applies equally well to computer monitors. Thus, a PNG viewer trying to achieve the maximum realism for the images it displays really needs to know what the room lighting conditions are, and adjust the gamma of the displayed image accordingly.

If asking the user about room lighting conditions is inappropriate or too difficult, just assume that the overall gamma (viewing_gamma as defined below) should be 1.0 or 1.25. That's all that most systems that implement gamma correction do.

What is a CRT's gamma?

All CRT displays have a power-law transfer characteristic with a gamma of about 2.5. This is due to the physical processes involved in controlling the electron beam in the electron gun, and has nothing to do with the phosphor.

An exception to this rule is fancy "calibrated" CRTs that have internal electronics to alter their transfer function. If you have one of these, you probably should believe what the manufacturer tells you its gamma is. But in all other cases, assuming 2.5 is likely to be pretty accurate.

There are various images around that purport to measure gamma, usually by comparing the intensity of an area containing alternating white and black with a series of areas of continuous gray of different intensity. These are usually not reliable. Test images that use a "checkerboard" pattern of black and white are the worst, because a single white pixel will be reproduced considerably darker than a large area of white. An image that uses alternating black and white horizontal lines (such as the

"gamma.png" test image at ftp://ftp.uu.net/graphics/png/images/suite/gamma.png) is much better, but even it may be inaccurate at high "picture" settings on some CRTs.

If you have a good photometer, you can measure the actual light output of a CRT as a function of input voltage and fit a power function to the measurements. However, note that this procedure is very sensitive to the CRT's black level adjustment, somewhat sensitive to its picture adjustment, and also affected by ambient

light. Furthermore, CRTs spread some light from bright areas of an image into nearby darker areas; a single bright spot against a black background may be seen to have a "halo". Your measuring technique will need to minimize the effects of this.

Because of the difficulty of measuring gamma, using either test images or measuring equipment, you're usually better off just assuming gamma is 2.5 rather than trying to measure it.

What is gamma correction?

A CRT has a gamma of 2.5, and we can't change that. To get an overall gamma of 1.0 (or somewhere near that) for an imaging system, we need to have at least one other component of the "image pipeline" that is nonlinear. If, in fact, there is only one nonlinear stage in addition to the CRT, then it's traditional to say that the CRT has a certain gamma, and that the other nonlinear stage provides "gamma correction" to compensate for the CRT. However, exactly where the "correction" is done depends on circumstance.

In all broadcast video systems, gamma correction is done in the camera. This choice was made in the days when television electronics were all analog, and a good gamma-correction circuit was expensive to build. The original NTSC video standard required cameras to have a transfer function with a gamma of 1/2.2, or about 0.45. Recently, a more complex two-part transfer function has been adopted [SMPTE-170M], but its behavior can be well approximated by a power function with a gamma of 0.5. When the resulting image is displayed on a CRT with a gamma of 2.5, the image on screen ends up with a gamma of about 1.25 relative to the original scene, which is appropriate for "dim surround" viewing.

These days, video signals are often digitized and stored in computer frame buffers. This works fine, but remember that gamma correction is "built into" the video signal, and so the digitized video has a gamma of about 0.5 relative to the original scene.

Computer rendering programs often produce linear samples. To display these correctly, intensity on the CRT must be directly proportional to the sample values in the frame buffer. This can be done with a special hardware lookup table between the frame

buffer and the CRT hardware. The lookup table (often called LUT) is loaded with a mapping that implements a power function with a gamma of 0.4, thus providing "gamma correction" for the CRT gamma.

Thus, gamma correction sometimes happens before the frame buffer, sometimes after. As long as images created in a particular environment are always displayed in that environment, everything is fine. But when people try to exchange images, differences in gamma correction conventions often result in images that seem far too bright and washed out, or far too dark and contrasty.

Gamma-encoded samples are good

So, is it better to do gamma correction before or after the frame buffer?

In an ideal world, sample values would be stored in floating point, there would be lots of precision, and it wouldn't really matter much. But in reality, we're always trying to store images in as few bits as we can.

If we decide to use samples that are linearly proportional to intensity, and do the gamma correction in the frame buffer LUT, it

turns out that we need to use at least 12 bits for each of red, green, and blue to have enough precision in intensity. With any less than that, we will sometimes see "contour bands" or "Mach bands" in the darker areas of the image, where two adjacent sample values are still far enough apart in intensity for the difference to be visible.

However, through an interesting coincidence, the human eye's subjective perception of brightness is related to the physical stimulation of light intensity in a manner that is very much like the power function used for gamma correction. If we apply gamma correction to measured (or calculated) light intensity before quantizing to an integer for storage in a frame buffer, we can get away with using many fewer bits to store the image. In fact, 8 bits per color is almost always sufficient to avoid contouring artifacts. This is because, since gamma correction is so closely related to human perception, we are assigning our 256 available sample codes to intensity values in a manner that approximates how visible those intensity changes are to the eye. Compared to a linear-sample image, we allocate fewer sample values to brighter parts of the tonal range and more sample values to the darker portions of the tonal range.

Thus, for the same apparent image quality, images using gamma-encoded sample values need only about two-thirds as many bits of storage as images using linear samples.

General gamma handling

When more than two nonlinear transfer functions are involved in the image pipeline, the term "gamma correction" becomes too vague. If we consider a pipeline that involves capturing (or calculating) an image, storing it in an image file, reading the file, and displaying the image on some sort of display screen, there are at least 5 places in the pipeline that could have nonlinear transfer functions. Let's give each a specific name for their characteristic gamma:

camera_gamma
: the characteristic of the image sensor

encoding_gamma
: the gamma of any transformation performed by the software writing the image file

decoding_gamma
: the gamma of any transformation performed by the software reading the image file

LUT_gamma
: the gamma of the frame buffer LUT, if present

CRT_gamma
: the gamma of the CRT, generally 2.5

In addition, let's add a few other names:

file_gamma
: the gamma of the image in the file, relative to the original scene. This is

```
file_gamma = camera_gamma * encoding_gamma
```

display_gamma
the gamma of the "display system" downstream of the frame buffer. This is

display_gamma = LUT_gamma * CRT_gamma

viewing_gamma
the overall gamma that we want to obtain to produce pleasing images --- generally 1.0 to 1.5.

The file_gamma value, as defined above, is what goes in the gAMA chunk in a PNG file. If file_gamma is not 1.0, we know that gamma correction has been done on the sample values in the file, and we could call them "gamma corrected" samples. However, since there can be so many different values of gamma in the image display chain, and some of them are not known at the time the image is

written, the samples are not really being "corrected" for a specific display condition. We are really using a power function in the process of encoding an intensity range into a small integer field, and so it is more correct to say "gamma encoded" samples instead of "gamma corrected" samples.

When displaying an image file, the image decoding program is responsible for making the overall gamma of the system equal to the desired viewing_gamma, by selecting the decoding_gamma appropriately. When displaying a PNG file, the gAMA chunk provides the file_gamma value. The display_gamma may be known for this machine, or it might be obtained from the system software, or the user might have to be asked what it is. The correct viewing_gamma depends on lighting conditions, and that will generally have to come from the user.

Ultimately, you should have

file_gamma * decoding_gamma * display_gamma = viewing_gamma

Some specific examples

In digital video systems, camera_gamma is about 0.5 by declaration of the various video standards documents. CRT_gamma is 2.5 as usual, while encoding_gamma, decoding_gamma, and LUT_gamma are all 1.0. As a result, viewing_gamma ends up being about 1.25.

On frame buffers that have hardware gamma correction tables, and that are calibrated to display linear samples correctly, display_gamma is 1.0.

Many workstations and X terminals and PC displays lack gamma correction lookup tables. Here, LUT_gamma is always 1.0, so display_gamma is 2.5.

On the Macintosh, there is a LUT. By default, it is loaded with a table whose gamma is about 0.72, giving a display_gamma (LUT and CRT combined) of about 1.8. Some Macs have a "Gamma" control panel that allows gamma to be changed to 1.0, 1.2, 1.4, 1.8, or 2.2. These settings load alternate LUTs that are designed to give a display_gamma that is equal to the label on the selected button. Thus, the "Gamma" control panel setting can be used directly as display_gamma in decoder calculations.

On recent SGI systems, there is a hardware gamma-correction table whose contents are controlled by the (privileged) "gamma" program. The gamma of the table is actually the reciprocal of the number that "gamma" prints, and it does not include the CRT gamma. To obtain the display_gamma, you need to find the SGI system gamma

(either by looking in a file, or asking the user) and then calculating


```
display_gamma = 2.5 / SGI_system_gamma
```

You will find SGI systems with the system gamma set to 1.0 and 2.2 (or higher), but the default when machines are shipped is 1.7.

A note about video gamma

The original NTSC video standards specified a simple power-law camera transfer function with a gamma of 1/2.2 or 0.45. This is not possible to implement exactly in analog hardware because the function has infinite slope at x=0, so all cameras deviated to some degree from this ideal. More recently, a new camera transfer function that is physically realizable has been accepted as a standard [SMPTE-170M]. It is

```
Vout = 4.5 * Vin                     if Vin < 0.018
Vout = 1.099 * (Vin^0.45) - 0.099    if Vin >= 0.018
```

where Vin and Vout are measured on a scale of 0 to 1. Although the exponent remains 0.45, the multiplication and subtraction change the shape of the transfer function, so it is no longer a pure power function. If you want to perform extremely precise calculations on video signals, you should use the expression above (or its inverse, as required).

However, PNG does not provide a way to specify that an image uses this exact transfer function; the gAMA chunk always assumes a pure power-law function. If we plot the two-part transfer function above along with the family of pure power functions, we find that a power function with a gamma of about 0.5 to 0.52 (not 0.45) most closely approximates the transfer function. Thus, when writing a PNG file with data obtained from digitizing the output of a modern video camera, the gAMA chunk should contain 0.5 or 0.52, not 0.45. The remaining difference between the true transfer function and the power function is insignificant for almost all purposes. (In fact, the alignment errors in most cameras are likely to be larger than the difference between these functions.) The designers of PNG deemed the simplicity and flexibility of a power-law definition of gAMA to be more important than being able to describe the SMPTE-170M transfer curve exactly.

The PAL and SECAM video standards specify a power-law camera transfer function with a gamma of 1/2.8 or 0.36 --- not the 1/2.2 of NTSC. However, this is too low in practice, so real cameras are likely to have their gamma set close to NTSC practice. Just guessing 0.45 or 0.5 is likely to give you viewable results, but if you want precise values you'll probably have to measure the particular camera.

Further reading

If you have access to the World Wide Web, read Charles Poynton's excellent "Gamma FAQ" [GAMMA-FAQ] for more information about gamma.

14. Appendix: Color Tutorial

(This appendix is not part of the formal PNG specification.)

About chromaticity

The cHRM chunk is used, together with the gAMA chunk, to convey precise color information so that a PNG image can be displayed or printed with better color fidelity than is possible without this information. The preceding chapters state how this information is encoded in a PNG image. This tutorial briefly outlines the underlying color theory for those who might not be familiar with it.

Note that displaying an image with incorrect gamma will produce much larger color errors than failing to use the chromaticity data. First be sure the monitor set-up and gamma correction are right, then worry about chromaticity.

The problem

The color of an object depends not only on the precise spectrum of light emitted or reflected from it, but also on the observer --- their species, what else they can see at the same time, even what they have recently looked at! Furthermore, two very different spectra can produce exactly the same color sensation. Color is not an objective property of real-world objects; it is a subjective, biological sensation. However, by making some simplifying assumptions (such as: we are talking about human vision) it is possible to produce a mathematical model of color and thereby obtain good color accuracy.

Device-dependent color

Display the same RGB data on three different monitors, side by side, and you will get a noticeably different color balance on each display. This is because each monitor emits a slightly different shade and intensity of red, green, and blue light. RGB is an example of a device-dependent color model --- the color you get depends on the device. This also means that a particular color --- represented as say RGB 87, 146, 116 on one monitor --- might have to be specified as RGB 98, 123, 104 on another to produce the same color.

Device-independent color

A full physical description of a color would require specifying the exact spectral power distribution of the light source. Fortunately, the human eye and brain are not so sensitive as to require exact reproduction of a spectrum. Mathematical, device-independent color models exist that describe fairly well how a particular color will be seen by humans. The most important device-independent color model, to which all others can be related, was developed by the International Lighting Committee (CIE, in French) and is called XYZ.

In XYZ, X is the sum of a weighted power distribution over the whole visible spectrum. So are Y and Z, each with different weights. Thus any arbitrary spectral power distribution is

condensed down to just three floating point numbers. The weights were derived from color matching experiments done on human subjects in the 1920s. CIE XYZ has been an International Standard since 1931, and it has a number of useful properties:

* two colors with the same XYZ values will look the same to humans
* two colors with different XYZ values will not look the same
* the Y value represents all the brightness information (luminance)
* the XYZ color of any object can be objectively measured

Color models based on XYZ have been used for many years by people who need accurate control of color --- lighting engineers for film and TV, paint and dyestuffs manufacturers, and so on. They are thus proven in industrial use. Accurate, device-independent color started to spread from high-end, specialized areas into the mainstream during the late 1980s and early 1990s, and PNG takes notice of that trend.

Calibrated, device-dependent color

Traditionally, image file formats have used uncalibrated, device-dependent color. If the precise details of the original display device are known, it becomes possible to convert the device-dependent colors of a particular image to device-independent ones. Making simplifying assumptions, such as working with CRTs (which are much easier than printers), all we need to know are the XYZ values of each primary color and the CRT_gamma.

So why does PNG not store images in XYZ instead of RGB? Well, two reasons. First, storing images in XYZ would require more bits of precision, which would make the files bigger. Second, all programs would have to convert the image data before viewing it. Whether calibrated or not, all variants of RGB are close enough that undemanding viewers can get by with simply displaying the data without color correction. By storing calibrated RGB, PNG

retains compatibility with existing programs that expect RGB data, yet provides enough information for conversion to XYZ in applications that need precise colors. Thus, we get the best of both worlds.

What are chromaticity and luminance?

Chromaticity is an objective measurement of the color of an object, leaving aside the brightness information. Chromaticity uses two parameters x and y, which are readily calculated from XYZ:

```
x = X / (X + Y + Z)
y = Y / (X + Y + Z)
```

XYZ colors having the same chromaticity values will appear to have the same hue but can vary in absolute brightness. Notice that x,y are dimensionless ratios, so they have the same values no matter what units we've used for X,Y,Z.

The Y value of an XYZ color is directly proportional to its absolute brightness and is called the luminance of the color. We can describe a color either by XYZ coordinates or by chromaticity x,y plus luminance Y. The XYZ form has the advantage that it is linearly related to (linear, gamma=1.0) RGB color spaces.

How are computer monitor colors described?

The "white point" of a monitor is the chromaticity x,y of the monitor's nominal white, that is, the color produced when R=G=B=maximum.

It's customary to specify monitor colors by giving the chromaticities of the individual phosphors R, G, and B, plus the white point. The white point allows one to infer the relative brightnesses of the three phosphors, which isn't determined by their chromaticities alone.

Note that the absolute brightness of the monitor is not specified. For computer graphics work, we generally don't care very much about absolute brightness levels. Instead of dealing with absolute XYZ values (in which X,Y,Z are expressed in physical units of radiated power, such as candelas per square meter), it is convenient to work in "relative XYZ" units, where the monitor's nominal white is taken to have a luminance (Y) of 1.0. Given this assumption, it's simple to compute XYZ coordinates for the monitor's white, red, green, and blue from their chromaticity values.

Why does cHRM use x,y rather than XYZ? Simply because that is how manufacturers print the information in their spec sheets! Usually, the first thing a program will do is convert the cHRM

chromaticities into relative XYZ space.

What can I do with it?

If a PNG file has the gAMA and cHRM chunks, the source_RGB values can be converted to XYZ. This lets you:

* do accurate grayscale conversion (just use the Y component)
* convert to RGB for your own monitor (to see the original colors)
* print the image in Level 2 PostScript with better color fidelity than a simple RGB to CMYK conversion could provide
* calculate an optimal color palette
* pass the image data to a color management system
* etc.

How do I convert from source_RGB to XYZ?

Make a few simplifying assumptions first, like the monitor really is jet black with no input and the guns don't interfere with one another. Then, given that you know the CIE XYZ values for each of red, green, and blue for a particular monitor, you put them into a matrix m:

```
        Xr Xg Xb
   m =  Yr Yg Yb
        Zr Zg Zb
```

Here we assume we are working with linear RGB floating point data in the range 0..1. If the gamma is not 1.0, make it so on the floating point data. Then convert source_RGB to XYZ by matrix multiplication:

```
   X     R
   Y = m G
   Z     B
```

In other words, X = Xr*R + Xg*G + Xb*B, and similarly for Y and Z. You can go the other way too:

```
   R      X
```

```
G = im Y
B      Z
```

where im is the inverse of the matrix m.

What is a gamut?

The gamut of a device is the subset of visible colors which that device can display. (It has nothing to do with gamma.) The gamut of an RGB device can be visualized as a polyhedron in XYZ space; the vertices correspond to the device's black, blue, red, green,

magenta, cyan, yellow and white.

Different devices have different gamuts, in other words one device will be able to display certain colors (usually highly saturated ones) that another device cannot. The gamut of a particular RGB device can be determined from its R, G, and B chromaticities and white point (the same values given in the cHRM chunk). The gamut of a color printer is more complex and can only be determined by measurement. However, printer gamuts are typically smaller than monitor gamuts, meaning that there can be many colors in a displayable image that cannot physically be printed.

Converting image data from one device to another generally results in gamut mismatches --- colors that cannot be represented exactly on the destination device. The process of making the colors fit, which can range from a simple clip to elaborate nonlinear scaling transformations, is termed gamut mapping. The aim is to produce a reasonable visual representation of the original image.

Further reading

References [COLOR-1] through [COLOR-5] provide more detail about color theory.

15. Appendix: Sample CRC Code

The following sample code represents a practical implementation of the CRC (Cyclic Redundancy Check) employed in PNG chunks. (See also ISO 3309 [ISO-3309] or ITU-T V.42 [ITU-V42] for a formal specification.)

The sample code is in the ANSI C programming language. Non C users may find it easier to read with these hints:

&

Bitwise AND operator.

^

Bitwise exclusive-OR operator. (Caution: elsewhere in this document, ^ represents exponentiation.)

>>

Bitwise right shift operator. When applied to an unsigned quantity, as here, right shift inserts zeroes at the left.

!

Logical NOT operator.

++

"n++" increments the variable n.

0xNNN
0x introduces a hexadecimal (base 16) constant. Suffix L indicates a long value (at least 32 bits).

```
/* Table of CRCs of all 8-bit messages. */
unsigned long crc_table[256];

/* Flag: has the table been computed? Initially false. */
int crc_table_computed = 0;

/* Make the table for a fast CRC. */
void make_crc_table(void)
{
  unsigned long c;
  int n, k;

  for (n = 0; n < 256; n++) {
    c = (unsigned long) n;
    for (k = 0; k < 8; k++) {
      if (c & 1)
        c = 0xedb88320L ^ (c >> 1);
      else
        c = c >> 1;
    }
    crc_table[n] = c;
  }
  crc_table_computed = 1;
}

/* Update a running CRC with the bytes buf[0..len-1]--the CRC
   should be initialized to all 1's, and the transmitted value
   is the 1's complement of the final running CRC (see the
   crc() routine below)). */

unsigned long update_crc(unsigned long crc, unsigned char *buf,
                         int len)
{
  unsigned long c = crc;
  int n;

  if (!crc_table_computed)
    make_crc_table();
  for (n = 0; n < len; n++) {
    c = crc_table[(c ^ buf[n]) & 0xff] ^ (c >> 8);
  }
  return c;
}
```



```
/* Return the CRC of the bytes buf[0..len-1]. */
unsigned long crc(unsigned char *buf, int len)
{
  return update_crc(0xffffffffL, buf, len) ^ 0xffffffffL;
}
```

16. Appendix: Online Resources

(This appendix is not part of the formal PNG specification.)

This appendix gives the locations of some Internet resources for PNG software developers. By the nature of the Internet, the list is incomplete and subject to change.

Archive sites

The latest released versions of this document and related information can always be found at the PNG FTP archive site, ftp://ftp.uu.net/graphics/png/. The PNG specification is available in several formats, including HTML, plain text, and PostScript.

Reference implementation and test images

A reference implementation in portable C is available from the PNG FTP archive site, ftp://ftp.uu.net/graphics/png/src/. The reference implementation is freely usable in all applications, including commercial applications.

Test images are available from ftp://ftp.uu.net/graphics/png/images/.

Electronic mail

The maintainers of the PNG specification can be contacted by e-mail at png-info@uunet.uu.net.

PNG home page

There is a World Wide Web home page for PNG at http://quest.jpl.nasa.gov/PNG/. This page is a central location for current information about PNG and PNG-related tools.

17. Appendix: Revision History

(This appendix is not part of the formal PNG specification.)

The PNG format has been frozen since the Ninth Draft of March 7, 1995, and all future changes are intended to be backwards compatible. The revisions since the Ninth Draft are simply clarifications, improvements in presentation, and additions of supporting material.

Changes since the Tenth Draft of 5 May, 1995

* Clarified meaning of a suggested-palette PLTE chunk in a truecolor image that uses transparency
* Clarified exact semantics of sBIT and allowed bit depth scaling procedures
* Clarified status of spaces in tEXt chunk keywords
* Distinguished private and public extension values in type and method fields
* Added a "Creation Time" tEXt keyword
* Macintosh representation of PNG specified

* Added discussion of security issues
* Added more extensive discussion of gamma and chromaticity handling, including tutorial appendixes
* Added a glossary
* Editing and reformatting

18. References

[COLOR-1]
Hall, Roy, Illumination and Color in Computer Generated Imagery. Springer-Verlag, New York, 1989. ISBN 0-387-96774-5.

[COLOR-2]
Kasson, J., and W. Plouffe, "An Analysis of Selected Computer Interchange Color Spaces", ACM Transactions on Graphics, vol 11 no 4 (1992), pp 373-405.

[COLOR-3]
Lilley, C., F. Lin, W.T. Hewitt, and T.L.J. Howard, Colour in Computer Graphics. CVCP, Sheffield, 1993. ISBN 1-85889-022-5. Also available from
<URL:http://info.mcc.ac.uk/CGU/ITTI/Col/colour_announce.html>

[COLOR-4]
Stone, M.C., W.B. Cowan, and J.C. Beatty, "Color gamut mapping and the printing of digital images", ACM Transactions on Graphics, vol 7 no 3 (1988), pp 249-292.

[COLOR-5]
Travis, David, Effective Color Displays --- Theory and Practice. Academic Press, London, 1991. ISBN 0-12-697690-2.

[GAMMA-FAQ]
Poynton, C., "Gamma FAQ".
<URL:http://www.inforamp.net/~poynton/Poynton-colour.html>

[ISO-3309]
International Organization for Standardization, "Information Processing Systems --- Data Communication High-Level Data Link Control Procedure --- Frame Structure", IS 3309, October 1984, 3rd Edition.

[ISO-8859]
International Organization for Standardization, "Information Processing --- 8-bit Single-Byte Coded Graphic Character Sets --- Part 1: Latin Alphabet No. 1", IS 8859-1, 1987.
Also see sample files at
ftp://ftp.uu.net/graphics/png/documents/iso_8859-1.*

[ITU-BT709]
International Telecommunications Union, "Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange", ITU-R Recommendation BT.709 (formerly CCIR Rec. 709), 1990.

[ITU-V42]
International Telecommunications Union, "Error-correcting Procedures for DCEs Using Asynchronous-to-Synchronous Conversion", ITU-T Recommendation V.42, 1994, Rev. 1.

[PAETH]
Paeth, A.W., "Image File Compression Made Easy", in Graphics Gems II, James Arvo, editor. Academic Press, San Diego, 1991. ISBN 0-12-064480-0.

[POSTSCRIPT]
Adobe Systems Incorporated, PostScript Language Reference Manual, 2nd edition. Addison-Wesley, Reading, 1990. ISBN 0-201-18127-4.

[PNG-EXTENSIONS]
PNG Group, "PNG Special-Purpose Public Chunks". Available in several formats from ftp://ftp.uu.net/graphics/png/documents/pngextensions.*

[RFC-1123]
Braden, R., Editor, "Requirements for Internet Hosts --- Application and Support", STD 3, RFC 1123, USC/Information Sciences Institute, October 1989.
<URL:ftp://ds.internic.net/rfc/rfc1123.txt>

[RFC-1521]
Borenstein, N., and N. Freed, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies", RFC 1521, Bellcore, Innosoft,

September 1993.
<URL:ftp://ds.internic.net/rfc/rfc1521.txt>

[RFC-1590]
Postel, J., "Media Type Registration Procedure", RFC 1590, USC/Information Sciences Institute, March 1994.
<URL:ftp://ds.internic.net/rfc/rfc1590.txt>

[RFC-1950]
Deutsch, P. and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, Aladdin Enterprises, May 1996.
<URL:ftp://ds.internic.net/rfc/rfc1950.txt>

[RFC-1951]
Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, Aladdin Enterprises, May 1996.
<URL:ftp://ds.internic.net/rfc/rfc1951.txt>

[SMPTE-170M]
Society of Motion Picture and Television Engineers, "Television --- Composite Analog Video Signal --- NTSC for Studio Applications", SMPTE-170M, 1994.

19. Credits

Editor

Thomas Boutell, boutell@boutell.com

Contributing Editor

Tom Lane, tgl@sss.pgh.pa.us

Authors

Authors' names are presented in alphabetical order.

* Mark Adler, madler@alumni.caltech.edu
* Thomas Boutell, boutell@boutell.com
* Christian Brunschen, cb@df.lth.se
* Adam M. Costello, amc@cs.wustl.edu
* Lee Daniel Crocker, lee@piclab.com
* Andreas Dilger, adilger@enel.ucalgary.ca
* Oliver Fromme, fromme@rz.tu-clausthal.de
* Jean-loup Gailly, gzip@prep.ai.mit.edu

* Chris Herborth, chrish@qnx.com
* Alex Jakulin, alex@hermes.si
* Neal Kettler, kettler@cs.colostate.edu
* Tom Lane, tgl@sss.pgh.pa.us
* Alexander Lehmann, alex@hal.rhein-main.de
* Chris Lilley, chris.lilley@mcc.ac.uk

* Dave Martindale, davem@cs.ubc.ca
* Owen Mortensen, ojm@csi.compuserve.com
* Robert P. Poole, lionboy@primenet.com
* Glenn Randers-Pehrson, glennrp@arl.mil or randeg@alumni.rpi.edu
* Greg Roelofs, newt@uchicago.edu
* Willem van Schaik, gwillem@ntuvax.ntu.ac.sg
* Guy Schalnat, schalnat@group42.com
* Paul Schmidt, pschmidt@photodex.com
* Tim Wegner, 71320.675@compuserve.com
* Jeremy Wohl, jeremy@cs.sunysb.edu

The authors wish to acknowledge the contributions of the Portable Network Graphics mailing list and the readers of comp.graphics.

Trademarks

GIF is a service mark of CompuServe Incorporated. IBM PC is a trademark of International Business Machines Corporation. Macintosh is a trademark of Apple Computer, Inc. Microsoft and MS-DOS are trademarks of Microsoft Corporation. PhotoCD is a trademark of Eastman Kodak Company. PostScript and TIFF are trademarks of Adobe Systems Incorporated. SGI is a trademark of Silicon Graphics, Inc. X Window System is a trademark of the Massachusetts Institute of Technology.

IESG Note

A disclaimer by the Internet Engineering Steering Group regarding intellectual property claims will be inserted here.

COPYRIGHT NOTICE

Copyright (c) 1996 by: Massachusetts Institute of Technology (MIT)

This W3C specification is being provided by the copyright holders under the following license. By obtaining, using and/or copying this specification, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, and distribute this specification for any purpose and without fee or royalty is hereby granted, provided that the full text of this NOTICE appears on ALL copies of the specification or portions thereof, including modifications, that you make.

THIS SPECIFICATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SPECIFICATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR

OTHER RIGHTS. COPYRIGHT HOLDERS WILL BEAR NO LIABILITY FOR ANY
USE OF THIS SPECIFICATION.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the specification without specific, written prior permission. Title to copyright in this specification and any associated documentation will at all times remain with copyright holders.

Security Considerations

Security issues are discussed in Security considerations (Section 8.5).

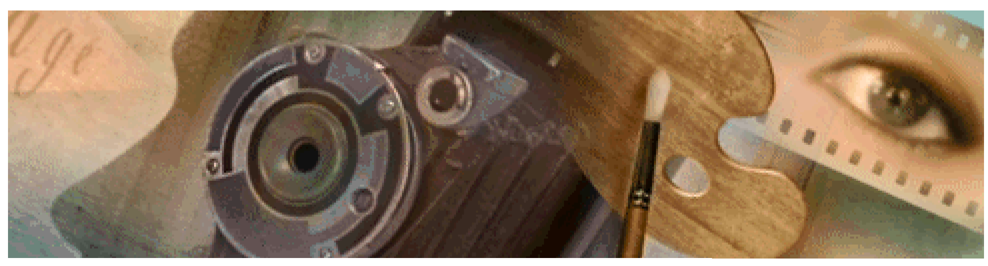
Author's Address

Thomas Boutell
PO Box 20837
Seattle, WA 98102

Phone: (206) 329-4969

EMail: boutell@boutell.com

End of PNG Specification



Adobe Photoshop™ 3.0

PLUG-IN TOOLKIT

Adobe Photoshop 3.0 Plug-in Toolkit

Copyright © 1991-95 Adobe Systems Incorporated All rights reserved.
Portions Copyright © 1990-91 Thomas Knoll

The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in this document. The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.
Adobe, Adobe Premiere, Adobe Photoshop, Adobe Illustrator, Adobe Type Manager, ATM and PostScript are trademarks of Adobe Systems Incorporated that may be registered in certain jurisdictions. Macintosh and Apple are registered trademarks of Apple Computer, Inc. MicrosoftMS, and MS-DOS are registered trademarks, and Windows is a trademark of Microsoft Corporation. All other products or name brands are trademarks of their respective holders.

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:

Date	Author	Description
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 6
How to Use This Toolkit 6
Plug-In Overview 6
Historical Note 6
Installation 7
Macintosh 7
Windows 7
Resources 7
'PiPL's 7
Definition 7
Structure 8
Notes 10
Types 10
General properties 11
Code Descriptor Properties 12
Filter specific properties 15
Format specific properties 18
Parser specific properties 20
'PiMI's 21
Execution 23
Macintosh 23
Windows 23

Callback Routines 24
TestAbort( ) 24
UpdateProgress( ) 24
ProcessEvent( ) 24
DisplayPixels( ) 25
GetPropertyProc( ) 27
Property keys 27
AdvanceStateProc ( ) 29
ColorServicesProc ( ) 29
Monitor Descriptions 31

Callback Suites 32
Buffer Suite 32
AllocateBuffer( ) 33
LockBuffer( ) 33
UnlockBuffer( ) 33
FreeBuffer( ) 33
BufferSpace( ) 33
Pseudo-Resource Suite 33
CountPIResources( ) 33
GetPIResource( ) 34
DeletePIResource( ) 34
AddPIResource( ) 34
Handle Suite 34
NewPIHandle ( ) 34
DisposePIHandle ( ) 34
GetPIHandleSize ( ) 34
SetPIHandleSize ( ) 34
LockPIHandle ( ) 35
UnlockPIHandle ( ) 35
RecoverSpaceProc ( ) 35

General Notes 35
Macintosh 35
Global Variables 35
Segmentation 35
About Boxes 36
Configuration 36
Windows 36
Configuration 36

About the Sample Plug-ins 37
Macintosh Version 37
Windows Version 37

Acquisition Modules 39
Basics 39
The AcquireRecord Structure 39
Record Fields 40
Calling Order 45
(1) acquireSelectorPrepare 45
(2) acquireSelectorStart 46
(3) acquireSelectorContinue 46
(4) acquireSelectorFinish 46
(5) acquireSelectorFinalize 47
State Machine 47
Error return values 49
Sample Plug-in 49
DummyScan 49

Export Modules 50
Basics 50
The ExportRecord Structure 50
Record Fields 51
Calling Order 55
(1) exportSelectorPrepare 55
(2) exportSelectorStart 55
(3) exportSelectorContinue 55
(4) exportSelectorFinish 55
State Machine 56
Error return values 56
Sample Plug-ins 57
DummyExport 57
HistoryExport 57
Paths to Illustrator 57

Filter Modules 58
Basics 58
The FilterRecord Structure 58
Record Fields 61
Calling Order 67
(1) filterSelectorParameters 67
(2) filterSelectorPrepare 68
(3) filterSelectorStart 68
(4) filterSelectorContinue 68
(5) filterSelectorFinish 68

State Machine 69
Error return values 70
Sample Plug-in 70
Dissolve 70

Image Format Modules 71
Basics 71
The FormatRecord Structure 71
Image Resources 72
Record Fields 73
Calling Sequences 77
(1) prepare 77
(2) start 78
(3) continue 78
(4) finish 79
Error return values 79
Sample Plug-in 79
Sample Format 79

Document File Formats 80
Image Resource Block 80
Path Resource Format 80
Photoshop 3.0 81
EPS 89
Filmstrip 91
TIFF 95

Load File Formats 97
Introduction 97
Arbitrary Map 98
Brushes 99
Color Table 101
Colors 102
Command Buttons 104
Curves 106
Duotone Options 108
Halftone Screens 110
Hue/Saturation 112
Ink Colors Setup 114
Custom Kernel 115
Levels 116
Monitor Setup 117
Replace Color/Color Range 118
Scratch Area 119
Selective Color 120
Separation Setup 121
Separation Tables 122
Transfer Function 124

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!)

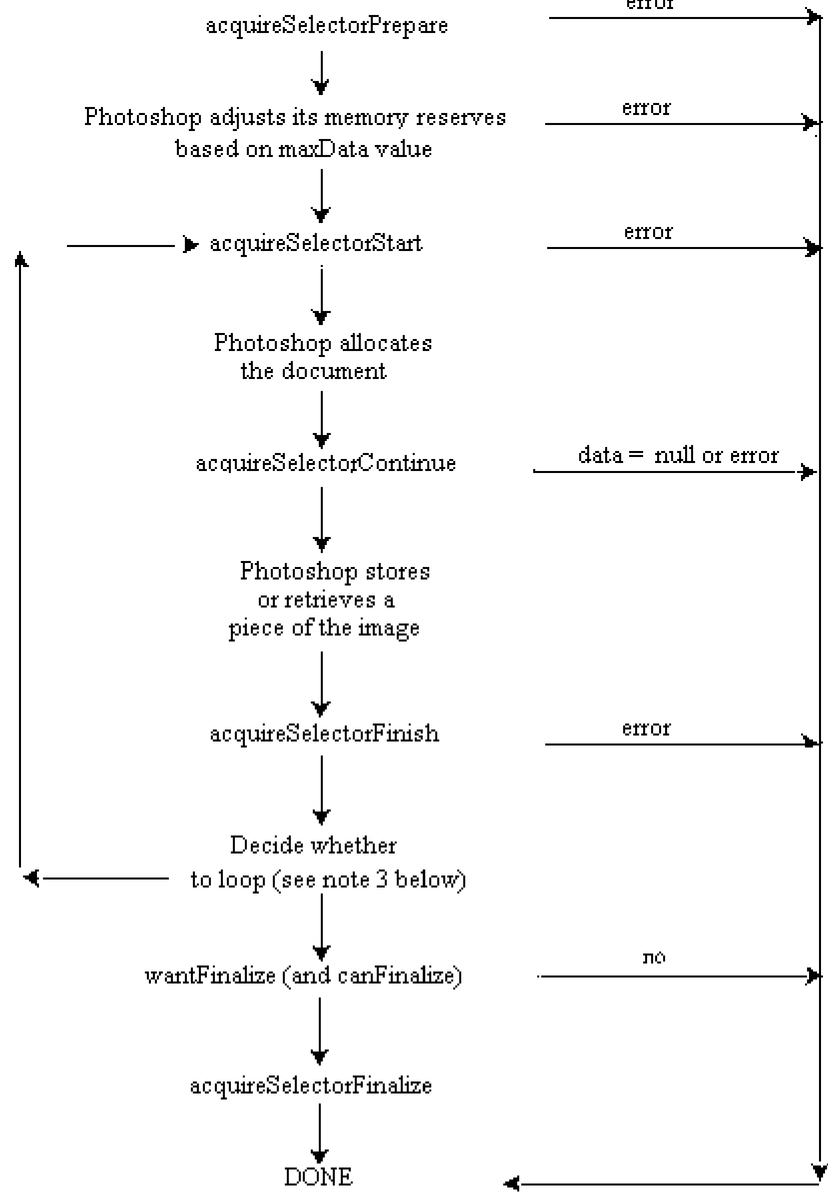
(4) acquireSelectorFinish
This call allows the plug-in to clean up after an image acquisition. This call is made if and only if the **acquireSelectorStart** routine returns without error (even if the **acquireSelectorContinue** routine returns an error).
Most plug-ins will at least need to free the buffer 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



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.

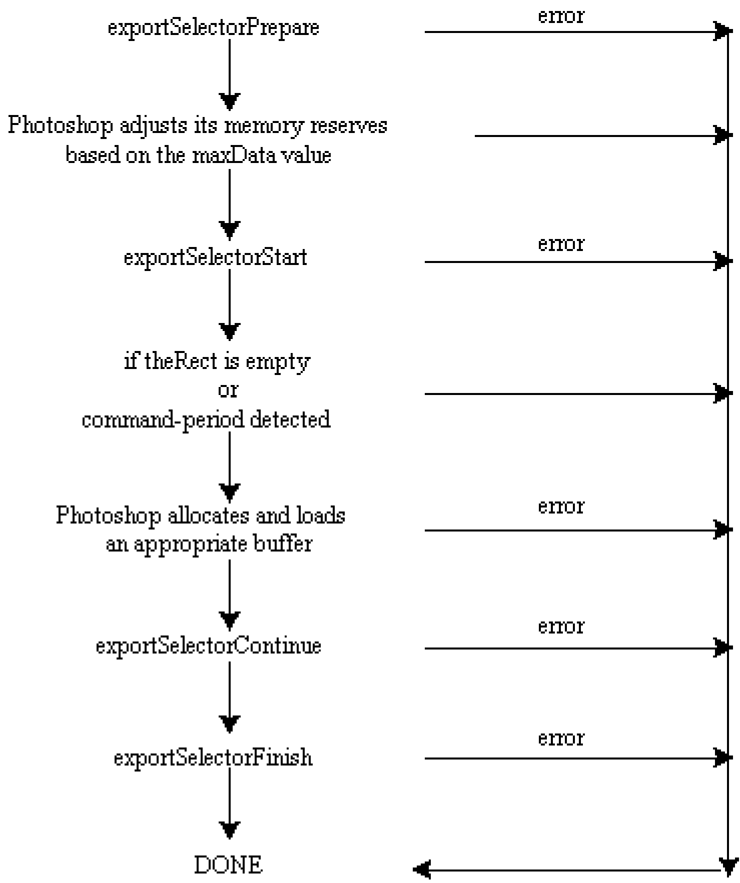
The requested image data is pointed to by data. If more than one plane has been requested (**loPlane** < > **hiPlane**), the data is interleaved. The offset from one row to the next is indicated by rowBytes. This is not necessarily equal to the width of **theRect** - there may be additional pad bytes 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



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.

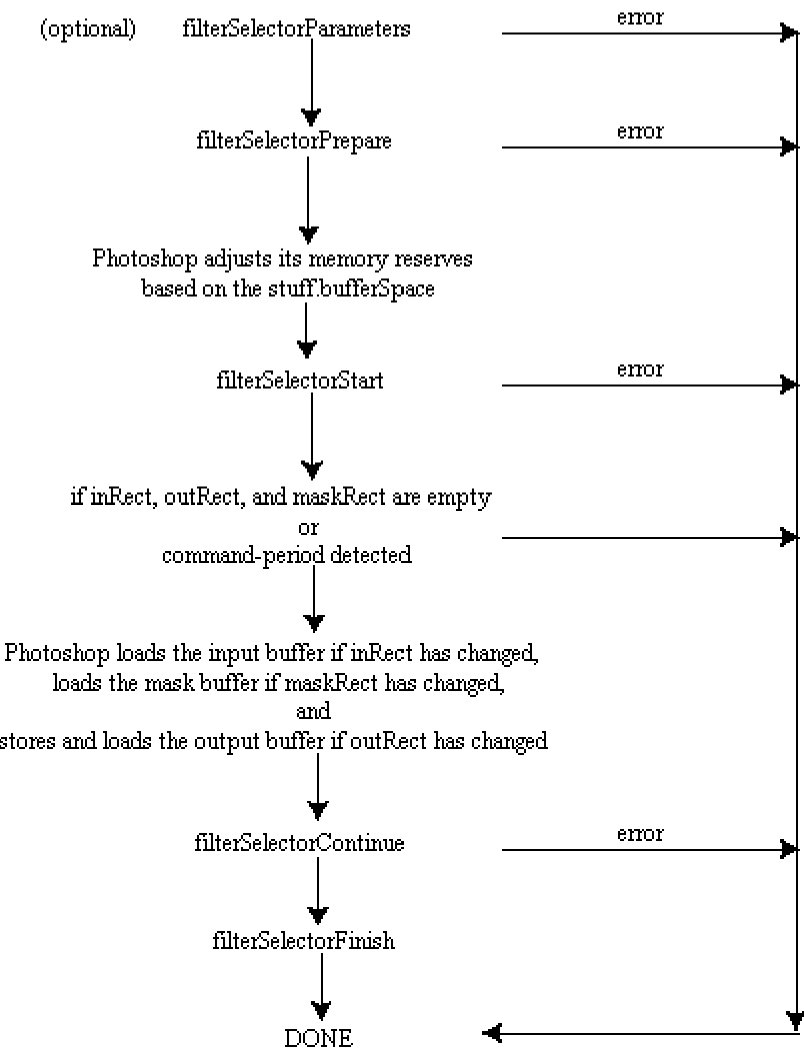
This routine should process the data pointed by **inData** and **outData** (and possibly **maskData**) and then update **inRect** and **outRect** (and **maskRect**, if using the selection mask) to request the next area of the image to process.

(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



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.

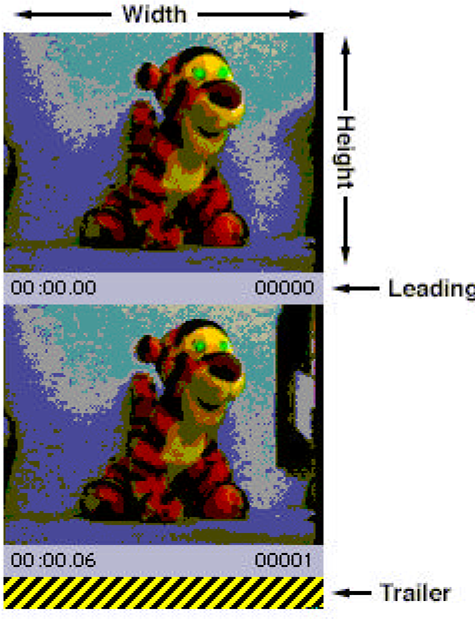
<binary/hex> is 1 if the data is in binary format, and 2 if the data is in hex format.

<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.

Adobe Photoshop_ 2.5 File Format

=A81992 Thomas Knoll
=A81993 Adobe Systems, Inc.

Photoshop 2.5 format was designed to be fast and easy to read and write, while solving a couple of problems with Photoshop 2.0 format:

(1) It is data fork only, thus cross-platform.

(2) It uses optional RLE compression, so simple images and mask channels can be saved much more compactly.

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 following sections describe the information stored in the file's data fork, 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 16 for Photoshop 2.5.

5. Rows (4 bytes)

The height of the image in pixels. Supported range is 1 to 30000 for Photoshop 2.5.

6. Columns (4 bytes)

The width of the image in pixels. Supported range is 1 to 30000 for Photoshop 2.5.

7. Depth (2 bytes)

The number of bits per channel. Supported values are 1 and 8 for Photoshop 2.5.

8. Mode (2 bytes)

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

9. Mode Data (4 byte 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 byte length + variable)

Contains a Photoshop 2.5 image resources block, the format of which is documented separately. Information contained in this block includes the image's resolution, and pen tool paths.

11. Reserved Data (4 byte length + variable)

Reserved for future use. Readers should skip this, and writers should write a zero length block.

12. Compression (2 bytes)

Two values are currently define: 0 =3D Raw Data, 1 =3D RLE compressed.

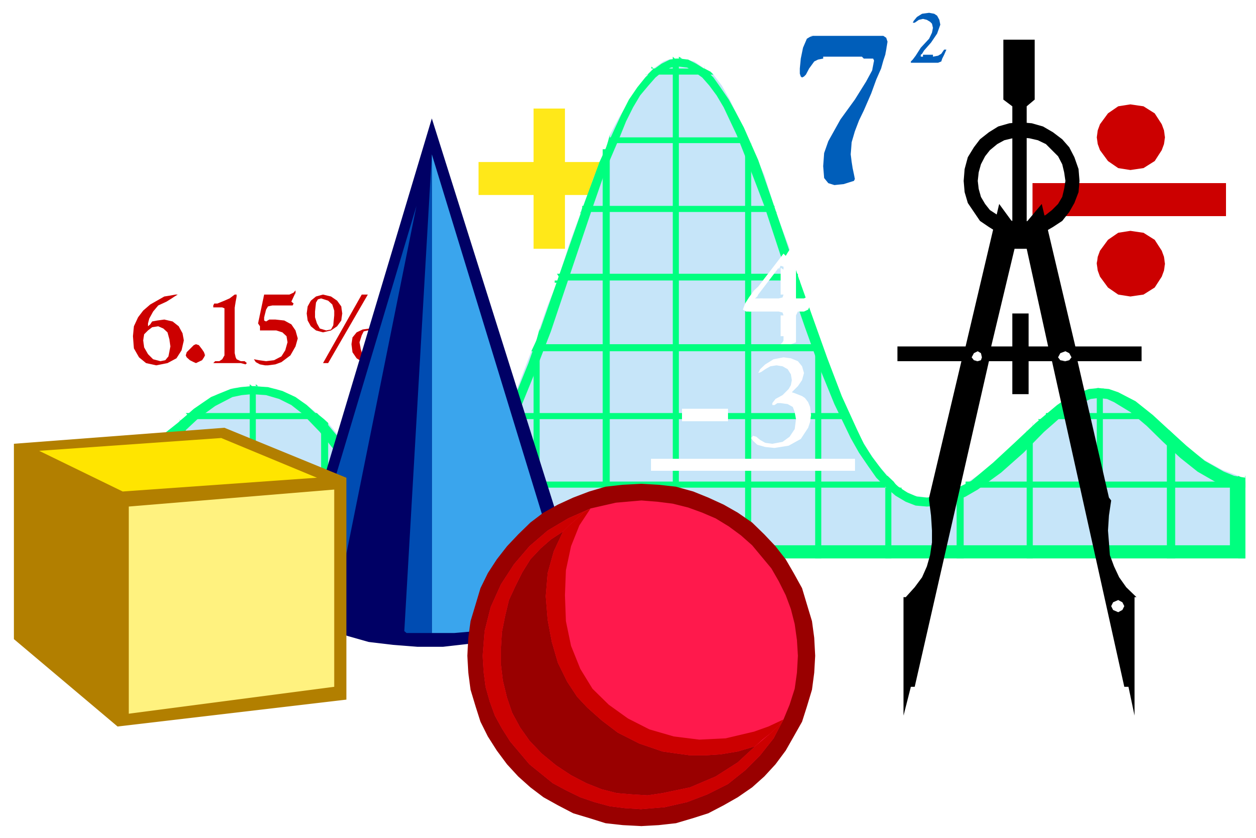
13. Image Data (variable)

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.

Gerber RS-274X Format



User's Guide

© Copyright 1998 Barco Graphics, Gent, Belgium, and Barco Gerber Systems, South Windsor, CT, USA.

All rights reserved. This material, information and instructions for use contained herein are the property of Barco Graphics N.V. and Barco Gerber Systems. The material, information and instructions are provided on an AS IS basis without warranty of any kind. There are no warranties granted or extended by this document. Furthermore Barco Graphics N.V. and Barco Gerber Systems do not warrant, guarantee or make any representations regarding the use, or the results of the use of the hardware, software or the information contained herein. Neither Barco Graphics nor Barco Gerber Systems shall be liable for any direct, indirect, consequential or incidental damages arising out of the use or inability to use the software or the information contained herein.

The information contained herein is subject to change without notice. Revisions may be issued from time to time to advise of such changes and/or additions.

No part of this document may be reproduced, stored in a data base or retrieval system, or published, in any form or in any way, electronically, mechanically, by print, photoprint, microfilm or any other means without prior written permission from Barco Graphics N.V. or Barco Gerber Systems.

This document supersedes all previous dated versions.

All product names are trademarks or registered trademarks of their respective owners. Correspondence regarding this publication should be forwarded to:

Engineering Services
Barco Gerber Systems Corporation
30 South Satellite Road
South Windsor, CT 06074

Printed in the USA

Published by
Barco Gerber Systems Corporation
Engineering Services Department

RS-274X Format User's Guide
Part Number 414-100-014 Rev C
September 21, 1998

Preface

The information in this Guide was published previously as the *Gerber Format Guide*, which documented *RS-274X format*, also called *Gerber Format*, for both vector and raster devices. Acknowledging that vector plotting is rapidly becoming an outdated technology, this Guide describes use of RS-274X only in raster applications and eliminates codes that pertain to vector applications. This does **not** imply that existing files that include codes and parameters not described in this edition of the Guide will not work. The intent of this Guide is to describe and document uses of RS-274X format.

Contents

INTRODUCTION 1

Who should use this Guide? 1
How to use this Guide 2
Related documentation 2
Additional copies of the Guide 2
Where to go for help 2

RULES AND GUIDELINES 3

File structure 3
  Data blocks 3
  Layers 3
  Data types 4
RS-274X parameters 4
  Directive parameters 5
  Image parameters 5
  Aperture parameters 5
  Layer-specific parameters 6
  Miscellaneous parameters 6
Standard RS-274D Codes and Coordinate Data 7
General File Preparation Guidelines 8
  RS-274X Parameter Guidelines 8
  Table 1 RS-274X Parameter Order of Entry 9
  RS-274D Data Guidelines 10
  Table 2 RS-274D Code Order of Entry 10
Sample Files 10
  Example 1 10
  Example 2 11

REFERENCE 15

RS-274X parameters 15
  AD - Aperture Definition 16
  AM - Aperture Macro 19
  AS - Axis Select 25
  FS - Format Statement 26
  IF - Include File 28
  IJ - Image Justify 29
  IN - Image Name 30
  IO - Image Offset 31
  IP - Image Polarity 32
  IR - Image Rotate 33
  KO - Knockout 34
  LN - Layer Name 36
  LP - Layer Polarity 37
  MI - Mirror Image 38
  MO - Mode 39
  OF - Offset 40
  PF - Plot Film 41

SF - Scale Factor 42
SR - Step and Repeat 43

STANDARD RS-274D CODES 45

D Codes 45
Table 4 D Codes 45
G Codes 46
Table 5 G Codes 46
Linear Interpolation (G01, G10, G11, G12) 47
Circular Interpolation (G02, G03, G74, G75) 47
Multiquadrant (360°) Circular Interpolation (G74, G75) 48
Polygon Area Fill (G36, G37) 49
M Codes 50

GLOSSARY 51

INDEX 53

Introduction

Gerber data is a simple, generic means of transferring printed circuit board information to a wide variety of devices that convert the electronic PCB data to artwork produced by a photoplotter. Virtually every PCB CAD system generates Gerber data because all photoplotters read it. It is a software structure consisting of X,Y coordinates supplemented by commands that define where the PCB image starts, what shape it will take, and where it ends. In addition to the coordinates, Gerber data contains aperture information, which defines the shapes and sizes of lines, holes, and other features.

Gerber Format, which is the format in which Gerber data is expressed, actually is a family of data formats that are subsets of EIA Standard RS-274D. *Extended Gerber Format*, which is also called *RS-274X*, provides enhancements that handle polygon fill codes, positive/negative image compositing, and custom apertures, and other features. RS-274X also encapsulates the aperture list in the header of the Gerber data file and therefore allows files to pass from one system to another without the need to re-input the aperture table. RS-274X produces a variety of Gerber data called *X data*.

RS-274X is a *superset* of the EIA Standard RS-274D format. RS-274X supports some of the parameter data codes (G codes) and aperture codes (D codes) contained in RS-274D, as well as codes referred to as *mass parameters*. Mass parameters are plot parameters that define characteristics that can affect an entire plot, or only specific parts of the plot, called *layers.* Mass parameters extend the capabilities of Gerber Format. Their presence makes the Gerber data *X data.*

RS-274X is maintained by Gerber Systems Corporation (GS), a leading supplier of CAD/CAM systems, large-area plotting systems, and precision cutting systems since 1965.

Who should use this Guide?

In order to use this Guide, you should have a fundamental understanding of PCB fabrication or PCB design and laser plotting concepts. This Guide is intended for use by:

- PCB designers preparing data for conversion to RS-274X
- PCB fabricators creating or using Gerber data files
- Developers of software applications using RS-274X data

How to use this Guide

You will find the following sections in this Guide:

Rules and Guidelines explain file content and structure and outlines rules and guidelines for creating a correct RS-274X file. It also contains a sample file.

Reference defines use and constraints on use of every RS-274X parameter and data code currently supported. Parameters and data codes are described separately. Both are presented in alphanumerical order.

You will also find a **Glossary** and **Index** at the end of the Guide.

Related documentation

This Guide assumes you are familiar with Electronic Industries Association EIA Standard RS-274D. You can obtain a copy of this standard from the Electronic Industries Association, Engineering Department, 2001 Eye Street NW, Washington, DC 20006 USA.

Where to go for help

Should you need assistance, contact the Baraco Gerber Systems Corporation Technical Assistance Center by telephone 8 a.m. to 5 p.m. (eastern time) at (860) 291-7016, by fax at (860) 291-7021.

Rules and Guidelines

This section provides background information, describes organization, and presents guidelines for use of RS-274X. For detailed descriptions of use of individual codes and parameters, see *Reference*, page 15.

File structure

An *RS-274X plot file* is a file consisting of RS-274X parameters and standard RS-274D codes which, when correctly interpreted, result in an image that may be displayed or plotted.

Data blocks

The file is composed of a number of *data blocks* containing parameters and codes. Each data block is delimited by an end-of-block character, typically an asterisk (*).

Each data block may contain one or more parameters or codes. For example:

```
X0Y0D02*
X50000Y0D01*
```

Layers

One or more data blocks may be grouped into a *layer* of information that describes part of a graphic image. In RS-274X context, a layer is a *named **information** component of the image* composed of one or more data blocks. Each layer may have characteristics, such as name, polarity, and interpolation mode, that differ from other layers of information . In addition, an individual layer may be "knocked out "of the surrounding graphic image, and may be repeated and/or rotated individually.

Note: A layer must not be confused with a PCB layer. A PCB layer has a physical definition and might be compared to a physical plane. An RS-274X layer is simply a group of data blocks that may be manipulated collectively and separately from other layers.

Data types

An RS-274X file may contain the following kinds of data appearing in the following general order:

1. **RS-274X Parameters**

   RS-274X parameters are also called *mass parameters* or *extended Gerber format*. The inclusion of these parameters in the file makes the plot file RS-274X, or *X data*, instead of standard RS-274D.

2. **Standard RS-274D Codes**

   Standard RS-274D codes were once called *word address format*. They consist of:

   - one-character **function codes** such as G codes, D codes, M codes, etc. Function codes were the *words* of the old terminology. They describe how coordinate data associated with them should be interpreted (such as linear or circular interpolation), how the imaging device should move (light source on or off), and more.
   - **coordinate data** define points to which the imaging device must move. The coordinate data represented the *address* of the old terminology. X,Y coordinate data describe linear positions. I, J coordinates define arcs.

RS-274X parameters

RS-274X parameters define characteristics that apply to an entire plot or to a single layer, depending on the parameter's position in the file and whether it generates a new information layer in the file (as, for example, layer-specific parameters do). RS-274X parameters consist of two alpha characters followed by one or more optional modifiers.

RS-274X parameters are delimited by a *parameter delimiter*, typically a percent (%) sign. Because parameters are also contained in a data block, they are also delimited by an end-of-block character. For example,

```
%FSLAX23Y23*%
```

This parameter is a Format Statement (FS) describing how the coordinate data in the file should be interpreted (in this case, 4.2 format for both X and Y coordinates). It is delimited by an end-of block character (*) as well as by a parameter delimiter (%).

RS-274X parameters may be grouped according to the scope of their function in the file. The groups should appear in the file in the following order:

1. **Directive parameters** control overall file processing.
2. **Image parameters** supply information about an entire image.
3. **Aperture parameters** describe the shape of lines and components throughout the file.
4. **Layer-specific parameters** describe processing of one or more data layers.
5. **Miscellaneous parameters** provide capabilities that do not fall into the above groups.

RS-274X parameters are generally placed at the beginning of the file in the order shown above. Certain parameters, such as the layer-specific parameters, may be embedded within the file.

The next sections describe each parameter type.

Directive parameters

Directive parameters control overall file processing. They include:

AS	Axis Select
FS	Format Statement
MI	Mirror Image
MO	Mode of units
OF	Offset
SF	Scale Factor

As a general rule, directive parameters should be placed at the beginning of the file. For simplicity sake, you should use each directive parameter only once in a file, although it is not illegal to use a directive parameter more than once. Each directive parameter controls processing until another like it is encountered.

When used more than once in a file, subsequent directive parameters may be embedded anywhere in the standard RS-274D code data or grouped with other layer-specific parameters.

Directive parameters do not generate a new layer.

Image parameters

Image parameters supply information about the entire (composite) image. Image parameters include:

IJ	Image Justify
IN	Image Name
IO	Image Offset
IP	Image Polarity
IR	Image Rotation
PF	Plotter Film

Image parameters should be used only once in a file and should be placed at the beginning of the file. If an image parameter occurs more than once in a file, the last one encountered will be the operative parameter.

Aperture parameters

Vector plotters control the width and shape of features by projecting light through a series of openings, or apertures, in a rotating wheel. Each position on the wheel is identified by a unique D code. When the D code appears in the data, the wheel rotates to the referenced position for exposure.

Unlike a vector device, a raster device has no apertures and therefore requires a description of the aperture geometry to create the required lines and other features. The aperture parameters provide the description.

The aperture parameters include:

AD	Aperture Description
AM	Aperture Macro

In general, aperture parameters apply to an entire file. An exception is an embedded AD parameter, which will generate a new layer if it redefines a D code previously used in the image data.

Note: Generating a new layer may result in unanticipated results because it causes certain RS-274D values (such as interpolation mode) to be reset.

The AM parameter describes a special aperture by using the following set of predefined aperture shapes to describe an aperture:

Circle
Line (vector)
Line (center)
Line (lower left)
Outline
Polygon
Moiré
Thermal

See the AM command description, page 19, for more information.

Layer-specific parameters

Layer-specific parameters supply information for the processing of one or more *information* layers (not to be confused with board layers). They may be used more than once in a file. Layer-specific parameters always generate a new layer and should be placed at the beginning of the new layer. If not repeated for a given layer, the previous layer-specific parameters remain in effect.

The layer-specific parameters include:

KO	Knockout
LN	Layer Name
LP	Layer Polarity
SR	Step and Repeat

Note: Generating a new layer may result in unanticipated results because it causes certain RS-274D values (such as interpolation mode) to be reset.

Miscellaneous parameters

There is a single miscellaneous parameter:

IF	Include File

The IF parameter is used to include (nest) external files in a file.

Standard RS-274D Codes and Coordinate Data

Standard RS-274D codes (D codes, G codes, M codes, etc.) specify how the coordinate data should be manipulated. Each code applies to coordinate data located in the same data block as the code (that is, between EOB characters). It also applies to coordinate data following it until another code of the same type is encountered, or until a new layer is generated. This continuing action is referred to as *modal*.

For example, G02 specifies clockwise, single-quadrant circular interpolation and is modal. All coordinate data following it will be considered clockwise arc data until another interpolation code is encountered, or until a new layer is generated. When a new layer is generated, interpolation will be reset to linear (G01).

Like parameters, standard RS-274D codes may be grouped according to function. They generally appear in the file in the following sequence:

1. **N codes** (sequence numbers) are similar to line numbers and may be assigned to data blocks to simplify organization. Sequence numbers may be 0 to 99999. N codes are not necessary.
2. **G codes** (general functions) specify how to interpolate and move to the coordinate locations following the code until changed or until a new layer is generated (modal).
3. **D codes** (plot functions) select and control tools, specify line type, etc.
4. **M codes** (miscellaneous functions) perform a variety of functions such as program stop and origin specification.

Standard RS-274D codes are described in detail starting on page 45.

Coordinate Data

Coordinate data includes:

- **X,Y data** define linear positions along the X and Y axes.
- **I,J data** define arcs.

For example,

```
X200Y200D02*
```

This data block directs the plotter to move in a positive direction to coordinate location 0.2,0.2 (assuming leading zeroes are omitted) with the light source off (tool up). Additional X,Y coordinate data positions following this code will also cause motion with the light source off until a different code is encountered.

Absolute versus relative data

Depending on the preceding FS parameter in the file, coordinate data may be defined either relative to the plot origin (that is, as absolute values) or relative to the last coordinate position (that is, as incremental data).

Numerical precision

Coordinate data may be expressed in inches or millimeters to ±6.6 decimal places (that is, up to six integer digits and six fractional digits). Unless preceded by a "-", the direction is assumed to be "+".

Axis assignment

The coordinate axes may be assigned to any physical plotter axes using the AS parameter, but typically the A plotter axis is assigned to X and the B axis to Y.

General File Preparation Guidelines

Follow these guidelines when preparing RS-274X data:

- Use data blocks in a way that organizes the file visually for easy reading.
- Enter all codes and parameters in upper case.
- Use file names that comply with the system file naming conventions. DOS and therefore Windows 3.1 files names are limited to eight characters. UNIX systems are case-sensitive.
- End every data block with an end-of-block character, typically *. For example,

```
X0000Y5000D02*
```

- Do not break a line within a block.
- End every file with an end-of-program code (M00 or M02).

RS-274X Parameter Guidelines

- Begin and end RS-274X parameter data with a parameter delimiter, typically %. The parameter delimiter must immediately follow the end-of-block without intervening spaces. For example,

```
%ASAXBY*%
```

- Parameters may be entered singly or grouped between delimiters, up to a maximum of 4096 characters between delimiters. A maximum of 80 characters between delimiters is recommended. Always consider readability. For example,

```
%SFA1.0B1.0*ASAXBY*%
```

- Line breaks are permitted between parameters to improve readability. For example,

```
%SFA1.0B1.0*
ASAXBY*%
```

- Use an explicit decimal point with all numerical values associated with a parameter. If the decimal point is omitted, an integer value is assumed.
- Express numerical values in the units defined by the MO code in the file (inches or millimeters).

- In general, enter RS-274X parameters at the beginning of the file (or at the beginning of the layer for layer-specific parameters). Enter them in the order shown in Table 1.

Note: When RS-274X parameters are embedded in the RS-274D data, all data preceding the parameter will be processed before the data blocks affected by the embedded parameters are interpreted.

Table 1 RS-274X Parameter Order of Entry

Parameter Required	Parameter Optional	Function	Comments	Default
	AS	Axis select	Single use recommended. When used more than once, enter these parameters at the beginning of a layer. These codes do not generate a new layer.	A=X, B=Y
FS		Format statement		
	MI	Mirror image		No mirror
	MO	Mode (inch or millimeter units)		Inch
	OF	Offset		A=0, B=0
	SF	Scale Factor		A=1.0, B=1.0
	IJ	Image Justify	Use only once at the beginning of the file.	No justification
IN		Image Name		
	IO	Image Offset		A=0, B=0
	IP	Image Polarity		Positive
	IR	Image Rotation		0
	PF	Plotter Film		
AD		Aperture Description	May be used singly or may be layer-specific. Enter these parameters at the beginning of the file or layer.	
	AM	Aperture Macro		
	LN	Layer Name		
	LP	Layer Polarity		Positive
	KO	Knockout		Off
	SR	Step and Repeat		A=1, B=1
	RO	Rotate		No rotation

RS-274D Data Guidelines

Follow these guidelines when preparing RS-274D data:

- Enter functions codes and coordinate data following the RS-274X parameters.
- Function codes apply to coordinate data in the same block as well as to subsequent coordinate data. They do not affect coordinate data preceding the block in which they occur.
- Enter function codes in the file in the order shown in Table 2.

Table 2 RS-274D Code Order of Entry

Code	Function	Comments
N	Sequence number	Optional
G	General functions: linear interpolation, circular interpolation, polygon area fill, etc.	Once encountered, remains in effect until countermanded.
D	Aperture or tool assignment; line/flash control	Once encountered, remains in effect until countermanded.
M	Miscellaneous function: program stop or end.	Every file must end with M00 or M02.

Sample Files

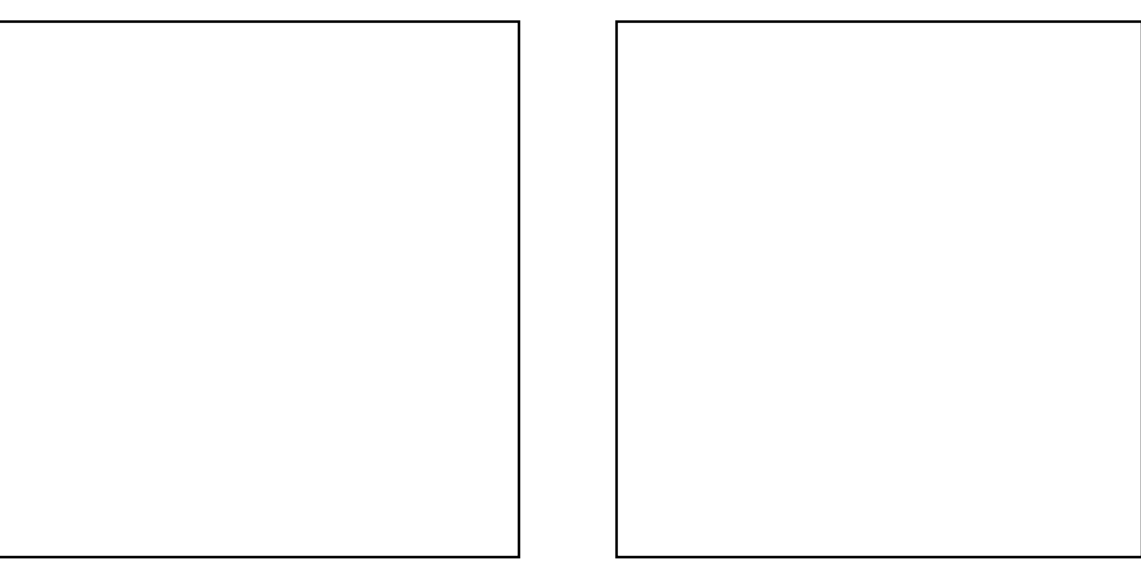
The examples on these pages illustrate the use of both mass parameters and standard RS-274D codes.

Example 1

Example 1 illustrates a single layer image.

Code	Comment
***G04 EXAMPLE 1: 2 BOXES**	
%FSLAX23Y23*%	Format statement - leading zeroes omitted, absolute coordinates, X2.3, Y2.3.
%MOIN*%	Set units to inches.
%OFA0B0*%	No offset
%SFA1.0B1.0*%	Scale factor is A1, B1
%ADD10C,0.010*%	Define aperture D-code 10 - 10 mil circle
%LNBOXES*%	Name layer “BOXES”.
G54D10	
X0Y0D02*X5000Y0D01*	
X5000Y5000D01*X0Y5000D01*X0Y0D01*	RS-274D data
X6000Y0*X11000Y0D01*	

X11000Y5000D01*X6000Y5000D01*	
X6000Y0D01*D02*	
M02*	End of data



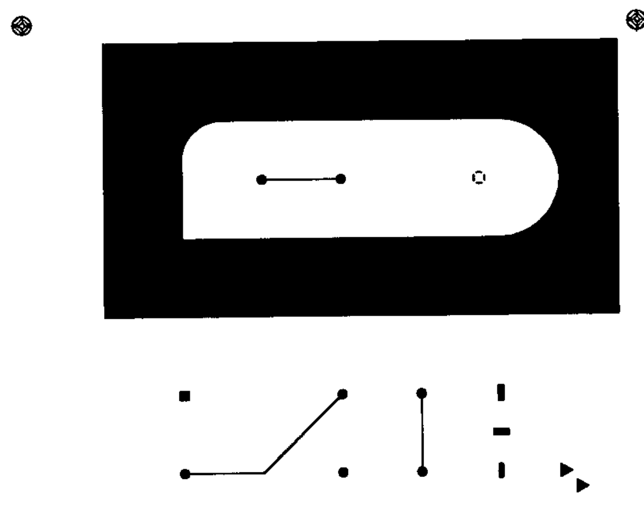
Example 2

Example 2 illustrates RS-274-X data.

%ASAXBY*	Axis Select, A=X, B=Y
FSLAX23Y23*	Format Statement, Leading zeros omitted, absolute data, 2 integer digits and 3 fractional digits
MIA0B0*	Mirror about the specified axis; 0=no, 1=yes
MOIN*	Mode inches
OFA0B0*	Offset 0
SFA1.0B1.0*%	Scale Factor
%IJALBL*	Image justify
INXTEST*	Image name
IOA0B0*	Image offset
IPPOS*	Image Polarity
IR0*%	Image Rotation
G04 Define Apertures*	Comment
%AMTARGET125*	Aperture Macro
6,0,0,0.125,.01,0.01,3,0.003,0.150,0*%	Moiré Description
%AMTHERMAL80*	Aperture Macro
7,0,0,0.080,0.055,0.0125,45*%	Thermal Description
%ADD10C,0.01*	Aperture Description, D10 is a circular aperture with 0.01” diameter
ADD11C,0.06*	Aperture Description, D11 is a circular aperture with 0.06” diameter
ADD12R,0.06X0.06*	Aperture Description, D12 is a rectangular aperture, 0.06” X 0.06”
ADD13R,0.04X0.100*	Aperture Description, D13 is a rectangular aperture, 0.04” X 0.100”
ADD14R,0.100X0.04*	Aperture Description, D14 is a rectangular aperture, 0.100” X 0.04”
ADD15O,0.04X0.100*	Aperture Description, D15 is a obround aperture, 0.04” X 0.100”
ADD16P,0.100X3*	Aperture Description, D16 is a 3 sided polygon 0.100” overall size
ADD17P,0.100X3*	Aperture Description, D17 is a 3 sided polygon 0.100” overall size
ADD18TARGET125*	Aperture Description, D18 is a special aperture called "TARGET”

Code	Description
ADD19THERMAL80*%	Aperture Description, D19 is a special aperture called "THERMAL"
%LNXTEST1*	Layer Name XTEST1
LPD*	Layer Polarity Dark
SRX1Y1I0J0*%	Step and Repeat set to 1 X 1 (Not Required)
G54D10*	Aperture select
G01X0Y250D02*	Linear move with light off
X0Y0D01*	Linear move with light on
X250Y0D01*	Linear move with light on
X1000Y1000D02*	Linear move with light off
X1500D01*	Linear move with light on
X2000Y1500*	Notice since D01 is modal it does not need to be repeated
X2500D02*	Notice since the X & Y commands are modal, Y is not repeated
Y1000D01*	Here, X is not repeated and uses its previous value of 2.500"
D02*	Light off no move
G54D11*	New aperture selected
G55X1000Y1000D03*	G55 prepares for flash It is not necessary. D03 is the flash command.
X2000D03*	Y value does not change
X2500D03*	This method reduces the size of the file
Y1500D03*	Here, X does not change from previous value
X2000D03*	Flash
G54D12*	New aperture select
X1000Y1500D03*	Move to (1.0, 1.5) and flash
G54D13*	New aperture select
X3000Y1500D03*	Move and flash
G54D14*	New aperture select
Y1250D03*	Move and flash
G54D15*	New aperture select
Y1000D03*	Move and flash
G54D10*	New aperture select
G01X3750Y1000D02*	Linear move, light off. Start point of the following arc command
G75*	Sets the mode to 360 degree circular interpolation
G03X3750Y1000I250J0D01*	Move from start point above to end point drawing a complete circle
G54D16*	New aperture select
G55X3400Y1000D03*	Flash
G54D17*	New aperture select
G55X3500Y900D03*	Flash
G54D10*	New aperture select
G36*	Start Polygon fill
G01X500Y2000D02*	
Y3750D01*	
X3750*	
Y2000*	
X500*	
X500Y2000D02*	
G37*	End Polygon fill
G54D18*	New aperture select
G55X0Y3875D03*	Flash
X3875Y3875D03*	Flash
%LNXTEST2*	Layer Name
LPC*%	Layer Polarity clear
G36*	Start Polygon fill
G01X1000Y2500D02*	
Y3000D01*	
G74*	Quadrant arc
G02X1250Y3250I250J0D01*	Clockwise arc move with radius .25"
G01X3000*	Complete 90 degree arc
G75*	360 degree arc mode

G02X3000Y2500I0J-375D01*	Clockwise arc move with radius .375"
G01X1000*	Linear move light on
X1000Y2500D02*	Linear move light off
G37*	End Polygon fill
%LNXTEST3*	Layer Name
LPD*%	Layer Polarity Dark
G54D10*	New aperture select
X1500Y2875D02*	
X2000D01*	
D02*	
G54D11*	
X1500Y2875D03*	
X2000D03*	
G54D19*	New aperture select
X2875Y2875D03*	
M02*	End of file



Reference

RS-274X parameters

This section describes every RS-274X parameter supported at time of publication. They are arranged in alphabetical order. Standard RS-274D code descriptions begin on page 45.

Each parameter description illustrates the parameter data block format, explains each parameter modifier, lists restrictions, and gives an example.

The data block format illustration uses the following notation conventions:

%Parameter code<required modifiers>[optional modifiers]*%

where:

Parameter code	is the 2-character code (AD, AM, FS, etc.)
<required modifiers>	must be entered to complete definition
[optional modifiers]	may be required depending on the required modifiers

The AD parameter is used to describe apertures (D codes) used in the RS-274X file. All apertures used in an RS-274X file must be described in terms of shape and size for the file to be interpreted correctly. The AD parameter must precede use of the associated aperture D-code. A definition remains in effect until redefined.

Two kinds of apertures may be used in an RS-274X file: *standard* apertures and *special* apertures.

Standard apertures

The AD parameter identifies standard apertures by D-code number and describes them in terms of shape (circular, rectangular, obround, or polygonal) and size (diameter if round, height and width if rectangular or obround, outside dimension and number of sides if polygonal). Apertures may be solid or open (that is, with a hole) and are always centered.

Special apertures

The AD parameter is also used to assign a D-code to a file containing an aperture description. The aperture description file may be a .mac file created by the AM (Aperture Macro) parameter or a .des file created by an Aperture Editor such as the Gerber GPC Aperture Editor. See the AM parameter description for further information on aperture macros.

AD parameter syntax rules

- Like other mass parameters, begin and end each parameter block with a parameter delimiter (typically %).
- Within the AD parameter block, separate each modifier by an X.
- Dimensions must be positive and will be rounded to the resolution of the output device.
- The various plotters and output devices may permit different D-code ranges, but the range must not exceed 10 to 999.

Data Block Format

%ADD<D-code number><aperture type>,<modifier>[X<modifer>]*%

where:

ADD — the AD parameter and D (for D-code)

<D-code number> — the D-code number being defined (10 - 999)

<aperture type>,<modifier>[X<modifier>] — the aperture descriptions. **<aperture type>** may be one of the following:

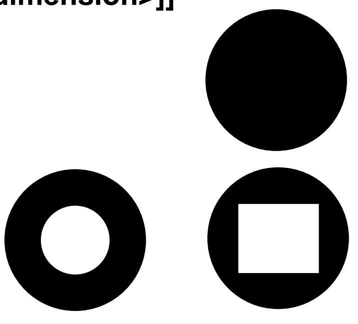
- a standard aperture (C, R, O, P, or T) with modifiers (see below). Modifiers depend on aperture type. Use **X** to separate each modifier. All dimensions are positive and will be rounded to the resolution of the output device.

- a file name containing the aperture description (.des file)
- an aperture macro name previously defined by the AM parameter (.mac file)

Note: Be sure to use the units (inches or millimeters) specified by the MO parameter for all modifiers.

Standard apertures:

C, **<outside diameter>[X<X-axis hole dimension >[X<Y-axis hole dimension>]]**

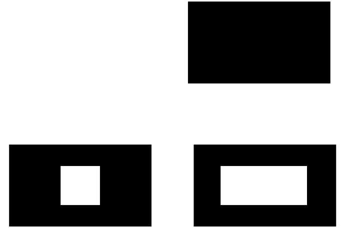


Circle. To define a solid aperture, enter only the diameter. To define a hole, enter one dimension for a round hole, two for a rectangle. The hole must fit within the aperture. For a square hole, X must equal Y. Both aperture and hole will be centered. For example,

%ADD10C,.05X0.025*%

D-code 10 is a .05 circle with a .025 round hole in the center.

R, **<X-axis dimension>X<Y-axis dimension>[X<X-axis hole dimension>X<Y-axis hole dimension>]**

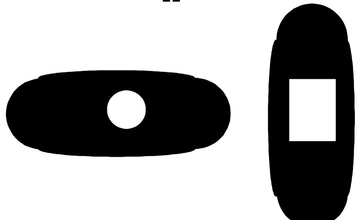


Rectangle or square. May be solid or open. If the X axis dimension equals the Y dimension, the aperture will be square. To define a solid aperture, enter only the X and Y dimensions; omit the hole dimensions. To define a hole, enter one dimension for a round hole, two for a rectangle. The hole must fit within the aperture. Both rectangle and hole will be centered. For example,

%ADD22R,0.020X0.040*%

D-code 22 is a .02 x .04 solid rectangle.

O, **<X-axis dimension>X<Y-axis dimension>[X<X-axis hole dimension>[X<Y-axis hole dimension>]]**

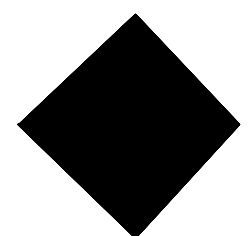


Obround (oval). May be solid or open. If the X dimension is larger than Y, the shape will be horizontal. If the X dimension is smaller than Y, the shape will be vertical. To define a solid aperture, enter only the X and Y dimensions; omit the hole dimensions. To define a hole, enter one hole dimension for a round hole, two for a rectangular or square hole. If open, the hole must fit within the aperture. For example,

%ADD22O,0.020X0.04X0.005X0.010*%

D-code 22 is a vertical obround .02 wide x .04 high with a .05 x .01 rectangular hole.

P, **<outside dimension>X<number of sides>[X<degrees of rotation>[X<X-axis hole dimension>X<Y-axis hole dimension>]]**



Regular polygon. May be solid or open. To define a solid aperture, enter only the outside dimension and number of sides (3 to 12). The first point is located on the X axis. May be rotated ±360° from the X-axis. If open, the hole must fit within the outside dimension. *Note: If you use the hole dimension modifiers, you must enter a rotation (even if it is 0).* For example,

%ADD17Diamond,.030X4X0.0*%

D-code 17 is a polygon within an outside dimension of .03, 4 sides, with no center hole.

Examples

%ADD10C,.025*%	Define D-code 10: 25 mil round
%ADD22R,.050X.050X.027*%	Define D-code 22: 50 mil square with 27 mil round hole
%ADD57O,.030X.040X.015*%	Define D-code 57: obround 30 x 40 mil with 15 mil round hole
%ADD30P,.016X6*%	Define D-code 30: polygon (hexagon), 16 mil outside dimension with 6 sides
%ADD15CIRC*%	Define D-code 15: a special aperture described by aperture macro CIRC defined previously by an aperture macro

The AM parameter is used to define named apertures (sometimes called *special apertures*) in *aperture macro* format consisting of building blocks called *primitives*. The named aperture macros may be used in AD parameter descriptions just like the standard apertures (that is, circle, rectangle, obround, polygon, and thermal). Every non-standard aperture must be described before the D-code associated with it occurs in the file.

Special apertures offer two advantages over standard apertures:

- They allow multiple shapes called primitives to be combined in a single aperture, which permits creation of unusual or complicated apertures.
- They need not be centered.
- Aperture macro modifiers may be variable. Variable modifiers are supplied by the AD parameter that references the aperture macro.
- An aperture macro variable may be a numerical function of another macro variable (+, -, etc.).

Aperture macro contents

An aperture macro contains the following elements:

- aperture macro name
- one or more of the seven aperture primitives, each identified by a primitive number (see Table 3 below for a description of the primitives)
- primitive modifiers that describe the primitive in terms of exposure, position, dimensions, etc.
- variable primitive modifiers to be supplied by the AD parameter
- optional embedded comment blocks
- numerical operators

AM parameter syntax rules

- Like other mass parameters, begin and end each parameter block with a parameter delimiter (typically %).
- Within the AM parameter block, separate each primitive and modifier group by an end-of-block character (typically *).
- Within each primitive group, separate modifiers by commas.
- Modifiers may be absolute values, such as 0, 1, 2, or 9.05, or they may be variable modifiers to be supplied by the AD parameter when it refers to the aperture macro.
- Identify variable modifiers to be supplied by the AD parameter as *$n* where *n* indicates the order in which the modifier is expected in the AD parameter. $1

would be the first variable modifier expected in the AD parameter, $2 the second, and so on, numbering sequentially from left to right. If an absolute value is entered instead of a variable, the variables shift right. For example, if an absolute value is entered for the first variable, the next variable becomes $1 even though it is the second modifier of the primitive.

- The interpretation of each modifier differs for each primitive. See Table 3 on the next page for a full explanation of aperture macro primitives and modifiers.
- Do not begin a variable primitive modifier with a minus sign (for example, -$1). To indicate negative, precede the variable with 0 (for example, **0-$1**).
- Start optional comment strings with a leading 0 (for example, ***0 THIS IS A COMMENT*).**
- Position and dimensions are expressed in the units specified by the MO parameter. Decimal points are permitted.
- Use only the following numerical operators with variable modifiers:

Operator	Function
+	add
-	subtract
/	divide
x	multiply
=	equate
n	numerical factor

- Make sure the aperture macro file name matches the aperture macro name and that it has a .mac extension.

Data Block Format

%AM<aperture macro name>*<primitive number>,<modifier$1>,<modifier$2>,[<...>]*[<primitive number>[<modifiers>]]*...*%

where:

AM	the AM parameter
<aperture macro name>*	the name to be used in the AD parameter
<primitive number>,<modifier$1>,<modifier$2>,<modifier$3>,...*	the primitive number with modifiers. The primitive number identifies the geometry (outline, polygon, etc.). The modifiers differ with the various primitive numbers. Use either actual values (for example, 0.050 for a width) or a variable placeholder (for example, $1 for exposure on/off).

Table 3 Aperture macro primitives

Primitive number	Description	Variable Modifiers	Description
1	**Circle**	$1	Exposure on/off 0 = OFF (laser off, no exposure) 1 = ON (laser on, image exposed) 2 = reverse current exposure state
		$2	Diameter
		$3	X center position
		$4	Y center position
2 or **20**	**Line (vector)**: a line defined by width, and beginning and end points. The line ends are rectangular.	$1	Exposure on/off 0 = OFF (laser off, no exposure) 1 = ON (laser on, image exposed) 2 = reverse current exposure state
		$2	Line width
		$3	X start point
		$4	Y start point
		$5	X end point
		$6	Y end point
		$7	Rotation in degrees (+ = counterclockwise, - = clockwise)
21	**Line (center)**: a centered rectangle defined by width, height, and center point. The end points are rectangular.	$1	Exposure on/off 0 = OFF (laser off, no exposure) 1 = ON (laser on, image exposed) 2 = reverse current exposure state
		$2	Rectangle width
		$3	Rectangle height
		$4	X center point
		$5	Y center point
		$6	Rotation in degrees (+ = counterclockwise, - = clockwise)

22	**Line (lower left)**: a rectangle defined by width, height, and the lower left point. The end points are rectangular.	$1	Exposure on/off 0 = OFF (laser off, no exposure) 1 = ON (laser on, image exposed) 2 = reverse current exposure state
	$4,$5 $3 $2	$2	Width
		$3	Height
		$4	X lower left point
		$5	Y lower left point
		$6	Rotation in degrees (+ = counterclockwise, - = clockwise)
3	End of file	none	Must be used to end .des files.
4	**Outline**: an open or closed shape defined by a start point, *n* additional points (up to 50), and the X,Y coordinates that define them. For a closed shape, the first and last points must be identical.	$1	Exposure on/off 0 = OFF (laser off, no exposure) 1 = ON (laser on, image exposed) 2 = reverse current exposure state
	$3,$4	$2	*n*, the number of points in the outline
		$3	X start point
		$4	Y start point
		$5	X point #1
		$6	Y point #1
		$7	X point #2
		$8, etc.	Y point #2. Continue as needed.
	$7,$8 $5,$6	$9 or the last number used	Rotation in degrees (+ = counterclockwise, - = clockwise)
5	**Polygon**: a closed, symmetrical, centered shape defined by n vertices (3 to 10 inclusive), a center point, diameter, and rotation.	$1	Exposure on/off 0 = OFF (laser off, no exposure) 1 = ON (laser on, image exposed) 2 = reverse current exposure state
	$3, $4	$2	number of vertices (integer)
	First Vertex	$3	X center point
		$4	Y center point
	$5	$5	Diameter
		$6	Rotation in degrees (+ = counterclockwise, - = clockwise)

6	**Moiré**: a cross hair centered on *n* concentric circles defined by the center point, outside diameter, line thickness, and gap between circles. $1, $2 $5 $8 $3	$1	X center point
		$2	Y center point
		$3	Outside diameter
		$4	Circle line thickness
		$5	Gap between circles
		$6	number of circles
		$7	Cross hair thickness
		$8	Cross hair length
		$9	Rotation in degrees (+ = counterclockwise, - = clockwise)
7	**Thermal**: a cross hair centered on a circle defined by outside and inside diameter. $4 $3	$1	X center point
		$2	Y center point
		$3	Outside diameter
		$4	Inside diameter
		$5	Cross hair thickness
		$6	Rotation in degrees (+ = counterclockwise, - = clockwise)

Example 1

%AMDONUT*1,1,$1,$2,$3*1,0,$4,$2,$3*% Define an aperture macro named DONUT consisting of two concentric circles:

1,1,$1,$2,$3 Circle (1), exposure on (1), diameter ($1), X center ($2), Y center ($3) all to be supplied by AD parameter

1,0,$4,$2,$3 Circle (1), exposure off (0), diameter ($4, different from first circle), X center and Y center ($2 and $3, same as first circle)

The AD parameter using this macro might look like the following:

%ADD32DONUT,0.100X0X0X0.080*% — Define D-code 32 to be aperture macro **DONUT**. The diameter of the first circle will be 0.100. The center of both circles will be at 0,0. The diameter of the second circle will be 0.080.

$1 = 0.100
$2 = 0
$3 = 0
$4 = 0.080

Example 2

%AMDONUT*1,1,$1,$2,$3*$1=$2+0.030*1,0,$1-$4,$2,$3*%

Define an aperture macro named **DONUT** consisting of two concentric circles with diameter of the second circle defined as a function of the diameter and center point of the first:

1,1,$1,$2,$3 — Circle (1), exposure on (1), diameter ($1), and center point X,Y ($2, $3) to be defined in the AD parameter

$1=$2+0.030 — Define a variable to be used to calculate the diameter of second circle to be a function of the diameter and center point X coordinate of the first.

1,0,$1-$4,$2,$3 — Circle (1), exposure off (0), diameter ($1-$4), and center point X,Y ($2, $3, same as the first circle).

The ADD parameter using this aperture macro might look like:

%ADD33DONUT,0.020X0X0X0.014*% — Define D-code 33 to be aperture macro **DONUT**. The diameter of the first circle 0.020. The center of both circles will be 0,0. The diameter of the second circle will be ((0 + 0.030) - 0.014).

Example 3

%AMDONUT*1,1,0.100X0X0*1,0,0.080X0X0*%

Define an aperture macro named **DONUT** consisting of two concentric circles, using primitive modifiers.

%ADD32DONUT*% — The resulting AD command only needs to reference the aperture macro name.

The AS parameter is used to assign any two data axes to the output device A or B axes.

Data Block Format

AS A<X or Y>B<X or Y>*

where:

A and B	are output device axes
X and Y	are data axes

Default

AXBY

Example

%ASAYBX*%	Assign the X axis data to the output device B axis and the Y axis data to the output device A axis.

The FS parameter is used to define the format of the input coordinate data and to define the allowable N, G, D, and M code lengths. It should be the first RS-274X parameter in the file. It is recommended that only one be used in the file. It is usually the first parameter.

The FS parameter allows you to specify the following format characteristics:

- Number of integer and decimal places in coordinate data (coordinate format)
- Zero omission (leading or trailing zeroes omitted)
- Absolute or incremental coordinate notation
- Sequence number (N-code) length
- General function code (G codes) length
- Draft code (D code) length
- Miscellaneous code (M code) length

Note: Decimal point programming is not supported.

Coordinate format

Coordinate format specifies how many integer and how many decimal places to expect in the coordinate data. For example, 2.3 format specifies two integer and three decimal places. A maximum of six integer and six decimal places may be specified (999999.999999). Different formats may be defined for the X and Y axes.

Zero omission

Zero omission compresses data by omitting either leading or trailing zeroes from coordinate values. Any given string of digits may be interpreted very differently depending on the zero omission specification. Coordinate format also affects how zero omission is interpreted.

Leading zero omission eliminates all zeroes that precede non-zero digits but retains following zeroes. For example, with 2.3 coordinate format, *15* would be interpreted as *0.015*.

Note: Use leading zero omission for NO ZEROES OMITTED files.

Trailing zero omission eliminates all zeroes following non-zero digits but retains preceding zeroes. For example, with 2.3 coordinate format, *15* would be interpreted as *15.000*.

Absolute or incremental notation

Coordinate values may be expressed as either absolute distances from a fixed 0,0 point or as relative distances from the preceding coordinate position.

RS-274D code lengths

The FS parameter can be used to specify length limits for the following standard RS-274D codes:

N Sequence number
G General function
D Plot function
M Miscellaneous function

These codes are described starting on page 45.

Data Block Format

%FS<L or T><A or I>[Nn][Gn]<Xn><Yn>[Dn][Mn]

where:

FS	The FS parameter
<L or T>	Use **L** to omit leading zeroes. Use **T** to omit trailing zeroes.
<A or I>	Use **A** for absolute coordinate values. Use **I** for incremental coordinate values.
[Nn], [Gn], [Dn], and [Mn]	Enter the code and an integer length limit, for example, **N2** to specify two-digit sequence codes.
<Xn> and <Yn>	Enter **X** or **Y** and the number of integer and decimal places in the coordinate data for each axis, for example, **X23** for X-axis data with two integer and three decimal places (99.999). 6.6 is maximum. The X and Y axes may have different values.

Example

%FSLAX25Y25*%	Coordinate data will have leading zeros omitted (L) and be expressed as absolute (A) positions with two integer and five decimal places in both axes (X25Y25).

The IF parameter is used to identify an external file to be included in the RS-274X file. The files referenced by the IF parameter will be interpreted exactly as if they were included at the point of reference in the RS-274X file. Included files may also contain IF parameters, up to 10 levels of nesting.

The IF parameter is often used to include an external aperture file containing AD and AM parameters that describe the apertures used in the RS-274X file, sometimes referred to as an "external" aperture list. The IF parameter can also be used to include external data files, which allows you to merge multiple data files. Included files simplify the creation of standard plot sequences such as title blocks and coupons.

Data Block Format

```
%IF<filename.ext>*%
```

Examples

%IFCOUPON3.GBR*%	Include file COUPON3.GBR.
%IFCIRCL.mac*%	Include aperture macro file CIRCL.mac.
%IFAPT004.des*%	Include aperture description file APT004.des.

The IJ parameter is used to override the absolute data coordinates for final placement of the image on the output device. The image may be centered or may be placed at an absolute position relative to the lower left of the platen.

Note: When centered, the pixel coordinates for the platen reside in the first quadrant (+X and +Y). X and Y are positive numbers, greater than zero and less than the platen size.

When more than one IJ parameter appear in the data, the final entry encountered is the one used.

Data Block Format

%IJ[A<parameter >B<parameter>][<offset>]*%

where:

IJ	the Image Justify parameter	
A	the plotter A axis justification	
<parameter>	**L**	left or lower justification (default)
	C	center justification
B	The plotter B axis justification	
<parameter>	**L**	left or lower justification (default)
	C	center justification
<offset>	the starting position offset relative to 0,0	

Default

None

Examples

%IJ*%	Left justify in X and lower justify in Y.
%IJAC*%	Center justify in X, lower justify in Y.
%IJACB.100*%	Center justify in X, offset .1 units in Y.
%IJALB.10*%	Left justify in X, offset .1 units in Y.
%IJB.100*%	Same as previous example.
%IJA1B1*%	Offset image 1 unit in X and Y.

IN
Image Name

The IN parameter is used to assign a name of up to 77 alphanumeric characters to the entire image of the RS-274X file. Information layers may also be named; see the LN parameter.

Data Block Format

%IN<character string>*%

where:

<character string> up to 77 alphanumeric characters except the asterisk (*).

Examples

%INSOLDERMASK*%

%INPANEL_1*%

The IO parameter is used to offset an image from the 0,0 point. The offset is expressed as an increment in the units defined by the MO parameter along the output device A and B axis. The AS parameter is used to correlate data axes with output device axes. The offset may be different for each axis and may be entered for a single axis.

Data Block Format

%IOA<±n>B<±n>*%

where:

IO	the Image Offset parameter
A<±n>	the offset along the output device A axis
B<±n>	the offset along the output device B axis

Default

%IOA0B0*%

Examples

%IOA1.0B1.5*%	Offset the image 1.0 units along the A axis and 1.5 units along the B axis from 0,0.
%IOB5.0*%	Offset the image 5.0 units along the B axis from 0,0.

IP
Image Polarity

The IP parameter is used to specify the positive or negative polarity of the entire file image. This *image polarity* differs from *layer polarity*, which is specified by the LP parameter and which applies only to one or more data layers of the entire image.

Data Block Format

%IP<POS or NEG>*%

where:

IP the IP parameter

<POS or NEG> Use **POS** for positive polarity, **NEG** for negative polarity.

Default

%IPPOS*%

Example

%IPNEG*% Output the entire image with negative polarity.

The IR parameter is used to rotate the entire image counterclockwise in 90° increments about the 0,0 coordinate. All apertures follow the rotation. If you do not use the IR parameter, 0° rotation is assumed.

Data Block Format

%IR<90 or 180 or 270>

where:

IR the IR parameter

<90 or 180 or 270> Enter the desired value.

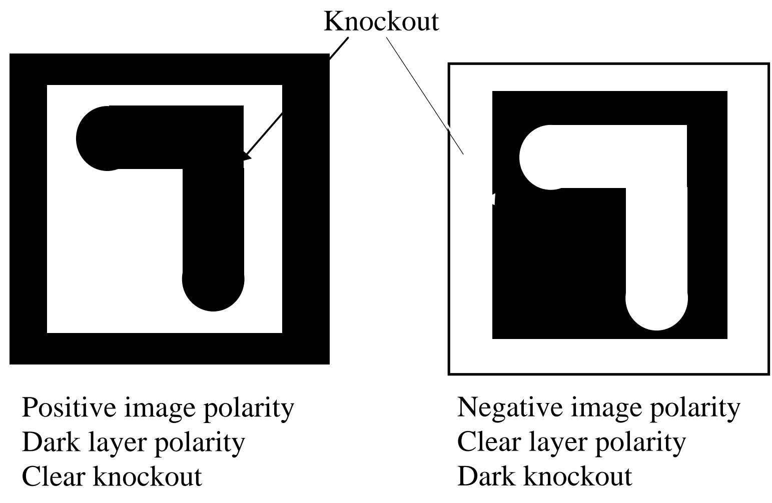
Default

%IR0*%

Example

%IR90*% Rotate the entire image 90° counterclockwise.

The KO parameter is used to specify that a rectangular region of the image will have the opposite polarity of the image or layer in which it occurs (either clear or dark), making it a "knockout" from the surrounding region.



Positive image polarity
Dark layer polarity
Clear knockout

Negative image polarity
Clear layer polarity
Dark knockout

KO is typically used to :

- set the initial background polarity of the final image
- set travel extent along the Y axis
- knock out a region around a component
- disable knockout on the current data layer

A knockout can be defined in one of two ways:

- lower left point and height and width
- border width around a component

When a knockout is defined around a component, the knockout is applied to all data following the KO parameter until the knockout is disabled. To disable a previously defined knockout, enter KO with no modifiers.

Data Block Format

%KO[C or D][X<coordinate>Y<coordinate>I<width>J<height>]or [K<border dimension>]

where:

KO the KO parameter

C or D Enter **C** for clear or **D** for dark. To create a knockout defined by the data extents, do not enter modifiers. To disable a previously enabled knockout, enter neither **C** nor **D**.

X<lower left X coordinate>Y<lower left Y coordinate>I<width>J<height>

Use this modifier to define the knockout by a lower left point, width, and height.

K<border dimension>	Use this modifier to define the knockout as a border around a component. Express the dimension in units specified by the MO parameter.

Examples

%KODX0I0I20J26*%	Create a dark knockout that extends from 0,0 to 20,26. This in effect sets the Y travel extent.
%KOCK.050*%	Create a clear knockout .050 units around all sides of the data that follows the KO parameter until the knockout is disabled.
%KOD*%	Make the region defined by the extents of the data following the command dark.
%KO*%	Disable a previously enabled knockout.

LN Layer Name

The LN parameter is used to assign a name of up to 77 alphanumeric characters to the information layer that follows the parameter in the RS-274X file. Entire image files may also be named; see the IN parameter.

Data Block Format

%LN<character string>*%

where:

<character string> up to 77 alphanumeric characters except the asterisk (*).

Examples

%LNSOLDERMASK*%

%LNINTERNAL_VCC*%

The LP parameter is used to specify the positive or negative polarity of the information layer or layers following it. This *layer polarity* differs from *image polarity*, which is specified by the IP parameter and which applies to the entire image. The layer polarity applies to all data following the LP parameter until another LP parameter is encountered.

Data Block Format

%LP<C or D>*%

where:

LP the IP parameter

<C or D> Use **C** for clear polarity, **D** for dark polarity.

Default

%LPD*%

Example

%LPC*% Make all succeeding data clear.

The MI parameter is used to turn mirror imaging either on or off. When on, all A- and/or B-axis data following the parameter will be mirrored (that is, inverted or multiplied by -1) until another MI command is used. Notice that mirroring A-axis data flips the image about the B axis. Mirroring B-axis data flips the image about the A axis.

Note: MI does not mirror special apertures.

The AS parameter is used to correlate the X and Y axes with the output device A and B axes.

Data Block Format

%MI[A<0 or 1>][B<0 or 1>]*%

where:

MI the MI parameter

A<0 or 1> To invert A-axis data (flipping the image about the B-axis), enter A1. To disable, enter A0.

B<0 or 1> To invert B-axis data (flipping the image about the A-axis), enter B1. To disable, enter B0.

Default

%MIA0B0*%

Example

%MIA0B1*% Disable mirroring of A-axis data. Invert B-axis data, flipping the image about the A-axis.

The MO parameter specifies that dimension data should be interpreted as inches or millimeters. Integer and decimal place format is specified by the FS parameter. Inches are assumed if units are not specified.

Data Block Format

%MO<IN or MM>*%

where:

MO	the MO parameter
<IN or MM>	Enter **IN** to specify inches. Enter **MM** to specify millimeters.

Default

%MOIN*%

Example

%MOIN*%	Dimension data will be expressed in inches.

The OF parameter is used to offset the final image up to ±99999.99999 units from the imaging device 0,0 point. The data may be offset along the imaging device A or B axis, or both. Values used with the OF parameter are expressed in units specified by the MO parameter, are always absolute, and are used primarily with absolute coordinate data. Incremental coordinate data may be offset simply by moving the imaging device to the desired offset position before starting the plot. The FS parameter specifies whether the data is absolute or incremental.

If an embedded FS parameter changes the format from absolute to incremental, the OF parameter value is saved and reinstated another FS parameter returns the format to absolute.

Data Block Format

%OF[A<±offset value>][B±offset value>]

where:

OF	the OF parameter
A±n	Offset along the A axis. Use 5.5 format.
B±n	Offset along the B axis. Use 5.5 format.

Default

%OFA0B0*%

Example

%OFA1.0B1.0*%	Offsets the plot 1 unit from 0,0 along both the A and B axes.

PF
Plot Film

The PF parameter is used to indicate to the operator the film (or other media) to be used to image the data file.

Data Block Format

%PF<name>*%

where:

PF	the PF parameter
<name>	up to 20 alphanumeric characters; asterisk (*) is an illegal character

The SF parameter is used to specify a scale factor of from 0.0001 to 999.99999 for the output device A- and/or B-axis data. The factor may be different for each axis. All data following the parameter will be multiplied by the factor until another SF parameter is encountered. The AS parameter is used to correlate the X and Y data axes with the imaging device A and B axes.

Data Block Format

%SF[A<factor>][B<factor>]*%

where:

SF	the SF parameter
A<factor>	the A-axis data factor
B<factor>	the B-axis data factor

Default

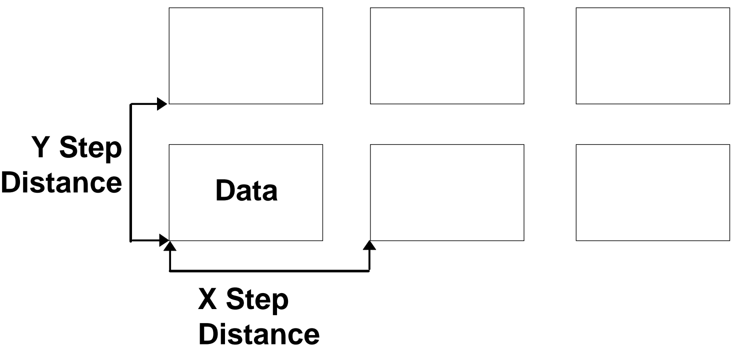
%A1B1*%

Example

%SFA.5B3*%	Multiply A-axis data by .5, B-axis data by 3.

The SR parameter is used to duplicate the data following the parameter a specific number of times (repeats) at a regular intervals (steps). The number of times the image is repeated and the space between repeats may be specified independently for X and Y data.

When entered without parameters, it is also used to disable a previous SR parameter.



Data Block Format

%SR[X<number of repeats>][Y<number of repeats>][I<X-axis step>][J<Y-axis step>]

where:

SR	the SR parameter
X<number of repeats>	the number of times the data will be repeated along the X-axis
Y<number of repeats>	the number of times the data will be repeated along the Y-axis
I<X-axis step distance>	the distance between the X-axis repeats
J<Y-axis step distance>	the distance between the Y-axis repeats

Default

%SRX1Y1I0J0*%

Examples

%SRX4I5.0J2*%	Repeat the image four times along the X axis with 5.0 units from one step to the next. The J modifier will be ignored because no Y repeats were specified.

%SRX2Y3I2.0J3*%	Repeat the image twice along the X axis and three times along the Y axis. X-axis repeats will be spaced 2.0 units apart. Y-axis repeats will be spaced three units apart.
%SR*%	Disable a previous SR parameter.

Standard RS-274D Codes

This section describes standard RS-274D codes (D codes, G codes, and M codes) that are applicable to raster output.

D Codes

D codes (draft codes) select apertures and determine whether the feature described should be imaged as a line or “flashed”. Table 4 lists supported D codes.

Table 4 D Codes

Code	Function	Comments
D01 (D1)	Draw line, exposure on	You cannot draw using an aperture defined by an aperture macro (AM parameter). These apertures can only be flashed.
D02 (D2)	Exposure off	
D03 (D3)	Flash aperture	D03 remains in effect until a new layer is encountered.
D10-D999	Select an aperture defined by an AD parameter.	

G Codes

G codes are general function codes. They specify how the coordinate data should be interpolated (linear or circular), turn the Polygon Area Fill feature on and off (see page 49 for more information on Polygon Area Fill), and can also be used to specify absolute or incremental format. Table 5 lists supported G codes.

Table 5 G Codes

Code	Function	Comments
G00	Move	Affected by Polygon Area Fill (see page 49.)
G01	Linear interpolation (1X scale)	See page 47.
G02	Clockwise circular interpolation	See page 47.
G03	Counterclockwise circular interpolation	
G04	Ignore data block	
G10	Linear interpolation (10X scale)	See page 47.
G11	Linear interpolation (0.1X scale)	
G12	Linear interpolation (0.01X scale)	
G36	Turn on Polygon Area Fill	See page 49.
G37	Turn off Polygon Area Fill	See page 49.
G54	Tool prepare	Usually precedes an aperture D-code
G70	Specify inches	See also MO parameter.
G71	Specify millimeters	See also MO parameter.
G74	Disable 360° circular interpolation (single quadrant)	See *Circular Interpolation*, page 47.
G75	Enable 360° circular interpolation (multiquadrant)	See *Circular Interpolation*, page 47.
G90	Specify absolute format	See also FS parameter.
G91	Specify incremental format	See also FS parameter.

Linear Interpolation (G01, G10, G11, G12)

Linear interpolation plots a straight line from the present position to the X,Y coordinate specified by the data block.

Data Block Format

G01 X±m.n Y±m.n Dnn

where:

G01 Specifies linear interpolation

X±m.n Y±m.n Defines the line end point

Dnn D-code (exposure on or off)

Circular Interpolation (G02, G03, G74, G75)

There are two types of circular interpolation: single quadrant (90°) and multiquadrant (360°). Single quadrant interpolation produces an arc. Multiquadrant interpolation can produce arcs that are larger than 90° and also circles.

Single Quadrant Circular Interpolation (G74)

G02 and G03 specify single quadrant (90°) circular interpolation; G74 disables it. Single quadrant circular interpolation plots an arc within one quadrant (90°). Single quadrant arcs must fit entirely within the quadrant in which they begin. A separate data block is required for each quadrant. A minimum of four data blocks is required to plot a circle.

Data Block Format

Gnn X±m.n Y±m.n Im.n Jm.n Dnn

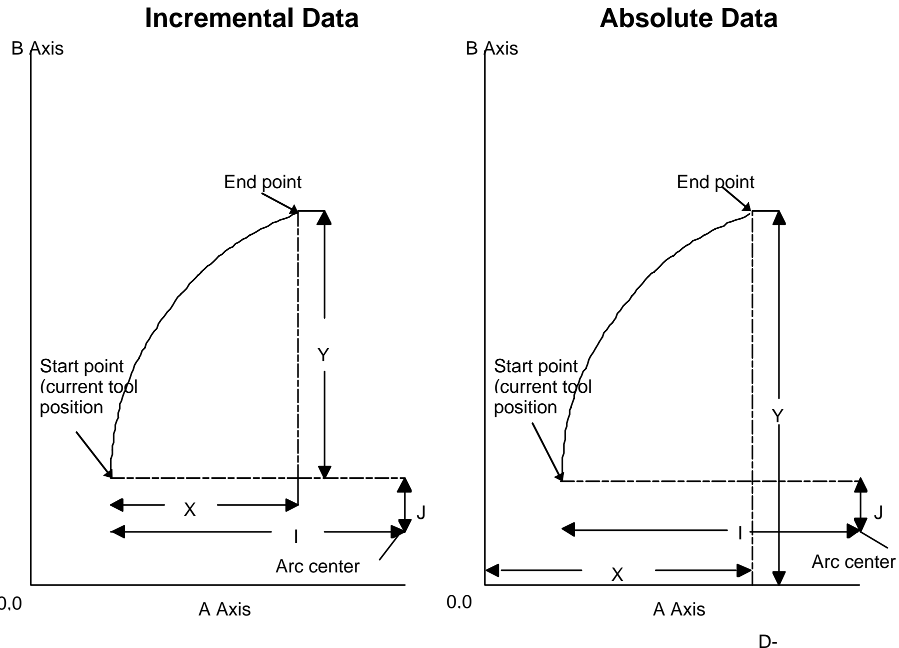
where:

Gnn G02 specifies clockwise circular interpolation
G03 specifies counterclockwise circular interpolation

X±m.n Y±m.n Defines the arc end point. These variables are in the format defined by the format statement (FS parameter). A sign is optional.

Im.n Jm.n Defines the incremental distance between the arc start point and the center measured parallel to the X and Y axes respectively. Notice that these numbers are unsigned values. The direction to the center is determined implicitly.

Dnn D-code (exposure on or off)



Multiquadrant (360°) Circular Interpolation (G74, G75)

A data block containing only G75 specifies 360° circular interpolation, which plots arcs in more than one quadrant using only one data block. Every block following a G75 code will be interpreted as 360° interpolation until a G74 is encountered. The I and J variables will be considered signed. If no sign is present, the circle will be in a positive direction from the start point.

A G74 code turns 360° multiquadrant circular interpolation off, reverting to single quadrant interpolation. To turn circular interpolation off and revert to linear interpolation, use G01.

Data Block Format

Gnn X±m.n Y±m.n ±Im.n ±Jm.n Dnn

where:

Gnn G74 turns off 360° circular interpolation
G75 turns on 360° circular interpolation

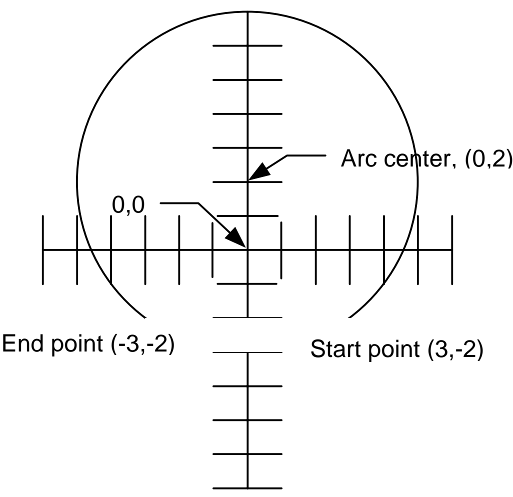
X±m.n Y±m.n Defines the arc end point

I±m.n J±m.n Defines the distance between the arc start point and the center measured parallel to the X and Y axes respectively. These variables are always incremental values in the format defined by the format statement (FS parameter). A sign is optional.

Dnn D-code (exposure on or off)

An example of multiquadrant interpolation is shown on the next page.

```
%FSLAX43Y43*%
G75*
G01X3000Y-2000D02*
G03X-3000Y-2000I-3000J4000D01
G01*
```

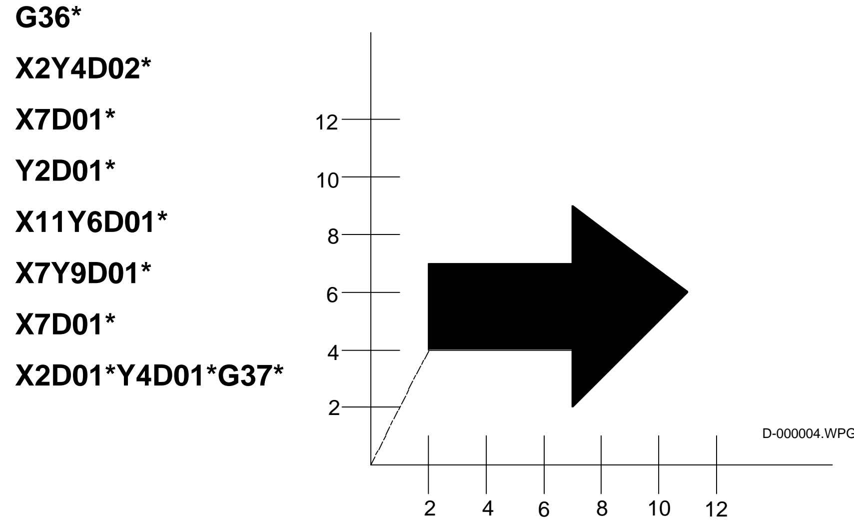


Polygon Area Fill (G36, G37)

G36 and G37 provide a more efficient means of filling closed polygons than stroke fill. When these codes are used, the filled area is defined simply by its closed outline. Stroke fill is an inefficient method of filling a polygon.

G36 turns on polygon area fill. G37 turn it off. There are no variables or apertures. Following a G36 and before G37, all lines drawn with D01 are considered edges of the polygon. D02 closes and fills the polygon.

```
G36*
X2Y4D02*
X7D01*
Y2D01*
X11Y6D01*
X7Y9D01*
X7D01*
X2D01*Y4D01*G37*
```



M Codes

M-codes identify the end of a file. Three M codes are commonly used:

M00	Program stop
M01	Optional stop
M02.	End of program

Glossary

ABSOLUTE POSITION: Position expressed as a distance from the 0,0 point in the data.

APERTURE: Previously, an opening in a wheel through which light passed to expose film. Currently a D code assignment and description of geometry that determines the shape of a feature.

APERTURE MACRO: A mass parameter that describes the geometry of a special aperture and assigns it to a D code.

APERTURE PARAMETER: A mass parameter (AD or AM) that assigns an aperture description to a D code.

CIRCULAR INTERPOLATION: Specifies that the data should be interpreted as arcs; may be single-quadrant or multiquadrant.

COMPOSITE IMAGE: The entire image, including all information layers.

COORDINATE DATA: X,Y position data that describes placement of features in the image.

D CODES: Draft (tool) RS-274D codes. They specify tool exposure action (line draw or flash).

DELIMITER: A character that indicates the beginning and end of a mass parameter.

DIRECTIVE PARAMETER: A mass parameter that controls overall file processing.

EXTENDED GERBER FORMAT: Gerber data that includes mass parameters.

FUNCTION CODES: G codes, D codes, M codes that are part of RS-274D.

G CODES: General function RS-274D codes. They specify interpolation, polygon area fill, etc.

GERBER DATA: Data expressed in Gerber Format.

GERBER FORMAT: A subset of RS-274D Word Address Format that is the universal plotter language; may also contain mass parameters, whose presence make it Extended Gerber Format, or RS-274X.

IMAGE PARAMETER: A parameter that supplies information about an entire image.

KNOCKOUT: A rectangular region about an information layer whose polarity is the opposite of the layer polarity.

LAYER: A named information component of Gerber data that may be treated as a unit, for example, rotated or repeated; has no relationship to a physical PCB layer.

LAYER-SPECIFIC PARAMETER: A mass parameter that applies to a single information layer (for example KO, LN, LP, and SR).

LINEAR INTERPOLATION: Specifies that the data should be interpreted as straight lines.

MASS PARAMETERS: Commands or codes that may be embedded in Gerber Data that specify how the data should be processed.

MULTI-QUADRANT INTERPOLATION: Specifies that the data should be interpreted as arcs that can extend into more than one quadrant, up to 360°).

NEGATIVE: An artwork in which the intended conductive pattern is transparent to light and the areas to be free from conductive material are opaque.

NUMERICAL PRECISION: The number of integer and decimal places used to express a number.

POLARITY: Describes whether the circuitry will be imaged as dark on a clear background (positive) or clear on a dark background (negative). Polarity may be applied to an entire image or to a single layer.

POLYGON AREA FILL: A feature that provides a more efficient means of creating solid (filled) polygons than stroke fill.

RELATIVE POSITION: Position expressed as a distance from the last position.

RS-274D: Electronics Industries Association (EIA) standard data format; a superset of Gerber Format.

RS-274X: Extended Gerber Format, that is, Gerber Format with mass parameters.

SINGLE QUADRANT INTERPOLATION: Specifies that the data should be interpreted as an arc that must fit entire within a single quadrant (90°).

STEP AND REPEAT: A method by which successive exposures of a single image are made to produce a multiple image production master.

STROKE FILL: An inefficient means of creating solid (filled) polygons by “painting” the area.

X DATA: Gerber data that includes mass parameters.

Index

A

absolute data coordinates
  override by IJ....29
Absolute notation....26
ABSOLUTE POSITION....51
absolute values....7
AD....6, 16
AM....6, 16, 19
APERTURE....51
Aperture Defintion....16
Aperture Description....6
Aperture Editor....16
Aperture Macro....6, 19, 51
APERTURE PARAMETER....51
Aperture parameters....5
apertures....5
  special....16
  standard....16
arc....47
arcs....7
AS....5, 25
Assistance....2
Axis assignment....8
Axis Select....5, 25

B

Bulletin Board Service (BBS)....2

C

center image....29
Circle....6, 17, 47, 48
circular interpolation....4, 47, 51
COMPOSITE IMAGE....51
coordinate data....4, 7, 51
  format....26

D

D codes....7, 16, 26, 45, 51
data block....3, 7, 8
  format....15
  maximum length....8
Data types....4
Decimal point programming....26
DELIMITER....51
DIRECTIVE PARAMETER....51
Directive parameters....5

E

Electronic Industries Association....2
end-of-block character....8
end-of-program code....8
English units
  specifying....39
Extended Gerber Format....1, 4, 51

F

file naming conventions....8
File structure....3
Format Statement....4, 5, 26
FS....5, 26
function codes....4, 51
functions codes....10

G

G codes....7, 26, 46, 51
G01, G10, G11, G12....47
G02, G03, G74, G75....47
G36, G37....49
G74....47
G74, G75....48
General File Preparation Guidelines....8
general function codes....46
general functions....7
Gerber data....1, 51
Gerber Format....1, 51
Gerber GPC Aperture Editor.....16
Glossary....51

I

I,J data....7
IF....6, 28
IJ....5, 29
Image Justify....5, 29
Image Name....5, 30
Image Offset....5, 31
IMAGE PARAMETER....5, 51
image placement....29

Image Polarity 5, 32
Image Rotation 5, 33
IN 5, 30
Inches
    specifying 39
Include File 6
incremental data 7
Incremental notation 26
IO 5, 31
IP 5, 32
IR 5, 33

J

justify image 29

K

Knockout 6, 34, 51
KO 6, 34

L

Layer 52
    generated by mass parameters 4, 6
Layer Name 6, 36
Layer Polarity 6, 37
Layers 3
LAYER-SPECIFIC PARAMETER 6, 52
Leading zero omission 26
Line (center) 6, 21
Line (lower left) 6, 22
Line (vector) 6, 21
Linear Interpolation 47, 52
LN 6, 36
LP 6, 37

M

M codes 7, 26, 50
M00 or M02 8
Mass parameters 4, 52
Metric units
    specifying 39
MI 5, 38
Millimeters
    specifying 39
Mirror Image 5, 38
Miscellaneous parameters 6
MO 5, 39
modal. 7
Mode 39
Mode of units 5
Moiré 6, 23
Multiquadrant (360°) Circular Interpolation ... 48, 52

N

N codes 7
Name
    image 30
    layer 36
NEGATIVE 52
NO ZEROES OMITTED 26
notation 26
Numerical precision 8, 52

O

Obround (oval) 17
OF 5, 40
Offset 5, 40
offset image 31
Order of entry
    RS-274D 10
    RS-274X 9
Ordering information 2
Outline 6, 22

P

Parameter delimiter 4, 8
Parameters
    placement of 5
PF 5, 41
Plot Film 5, 41
polarity 34, 52
    image 32
    layer 37
Polygon 6, 22
polygon area fill 1, 49, 52
primitives 19

R

raster device 5
Rectangle or square 17
Regular polygon 18
RELATIVE POSITION 52
Rotate
    image 33
RS-274D 1, 2, 7, 52
    code length 27
    Data Guidelines 10
    order of entry 10
RS-274X 1, 4, 8, 15, 52
    defaults 9
    order of entry 9
    Parameter Guidelines 8
    position in file 9
    required and optional 9

S

Sample Files 10
Scale Factor 5, 42
Sequence number 7, 26
SF 5, 42
Single Quadrant Circular Interpolation 47, 52
Special apertures 16, 19
SR 6, 43
Standard apertures 16, 17
Standard RS-274D Codes 4, 45

Step and Repeat 6, 43, 52
STROKE FILL 52

T

Technical Assistance Center 2
Thermal...6, 23
Trailing zero omission 26

U

Units..39
upper case
    required for entry..8

V

Vector plotters 5

W

Web Page 2
word address format.. 4

X

X data... 1, 52
X,Y data 7

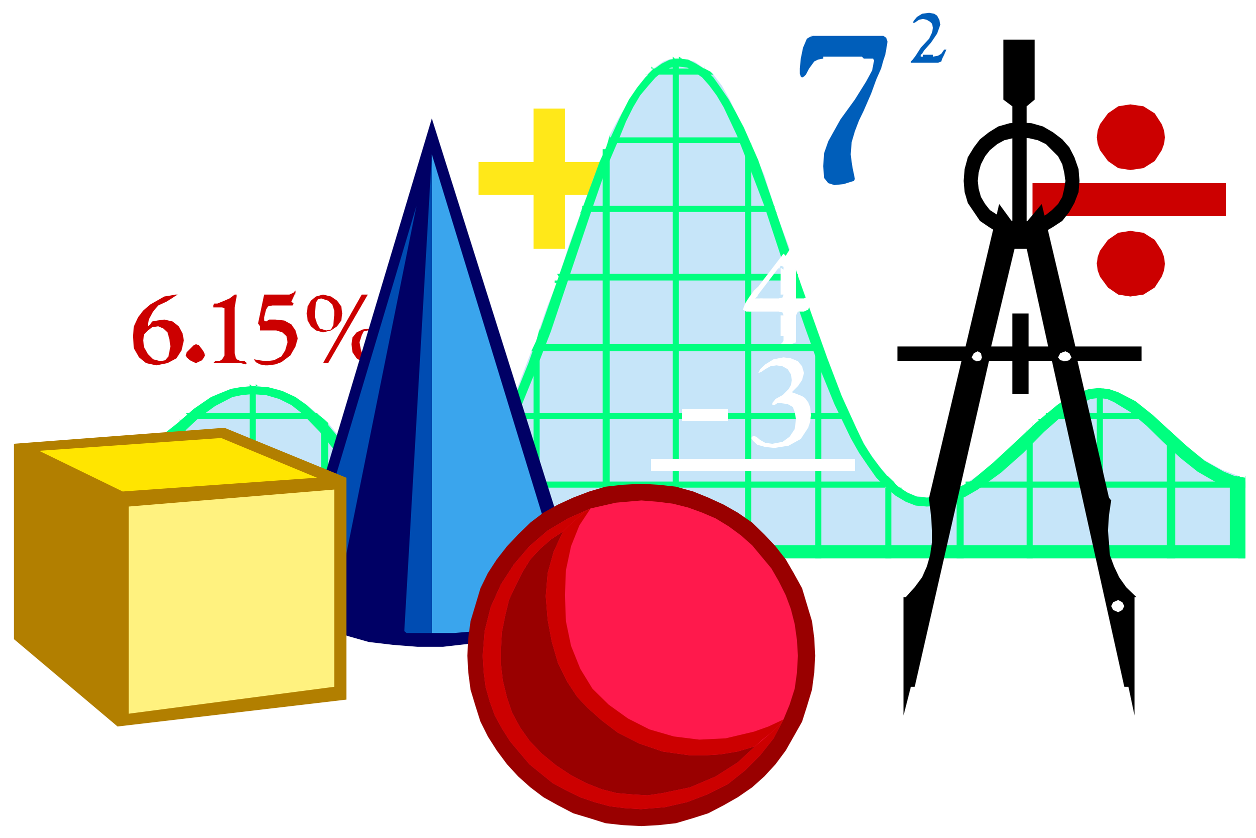
Z

Zero omission... 26

GERBER SYSTEMS

Part Number 414 100 014 C September 21, 1998

Gerber RS-274X Format



User's Guide

© Copyright 1998 Barco Graphics, Gent, Belgium, and Barco Gerber Systems, South Windsor, CT, USA.

All rights reserved. This material, information and instructions for use contained herein are the property of Barco Graphics N.V. and Barco Gerber Systems. The material, information and instructions are provided on an AS IS basis without warranty of any kind. There are no warranties granted or extended by this document. Furthermore Barco Graphics N.V. and Barco Gerber Systems do not warrant, guarantee or make any representations regarding the use, or the results of the use of the hardware, software or the information contained herein. Neither Barco Graphics nor Barco Gerber Systems shall be liable for any direct, indirect, consequential or incidental damages arising out of the use or inability to use the software or the information contained herein.

The information contained herein is subject to change without notice. Revisions may be issued from time to time to advise of such changes and/or additions.

No part of this document may be reproduced, stored in a data base or retrieval system, or published, in any form or in any way, electronically, mechanically, by print, photoprint, microfilm or any other means without prior written permission from Barco Graphics N.V. or Barco Gerber Systems.

This document supersedes all previous dated versions.

All product names are trademarks or registered trademarks of their respective owners. Correspondence regarding this publication should be forwarded to:

Engineering Services
Barco Gerber Systems Corporation
30 South Satellite Road
South Windsor, CT 06074

Printed in the USA

Published by
Barco Gerber Systems Corporation
Engineering Services Department

RS-274X Format User's Guide
Part Number 414-100-014 Rev C
September 21, 1998

Preface

The information in this Guide was published previously as the *Gerber Format Guide*, which documented *RS-274X format*, also called *Gerber Format*, for both vector and raster devices. Acknowledging that vector plotting is rapidly becoming an outdated technology, this Guide describes use of RS-274X only in raster applications and eliminates codes that pertain to vector applications. This does **not** imply that existing files that include codes and parameters not described in this edition of the Guide will not work. The intent of this Guide is to describe and document uses of RS-274X format.

Contents

INTRODUCTION **1**

Who should use this Guide? 1
How to use this Guide 2
Related documentation 2
Additional copies of the Guide 2
Where to go for help 2

RULES AND GUIDELINES **3**

File structure 3
  Data blocks 3
  Layers 3
  Data types 4
RS-274X parameters 4
  Directive parameters 5
  Image parameters 5
  Aperture parameters 5
  Layer-specific parameters 6
  Miscellaneous parameters 6
Standard RS-274D Codes and Coordinate Data 7
General File Preparation Guidelines 8
  RS-274X Parameter Guidelines 8
  Table 1 RS-274X Parameter Order of Entry 9
  RS-274D Data Guidelines 10
  Table 2 RS-274D Code Order of Entry 10
Sample Files 10
  Example 1 10
  Example 2 11

REFERENCE **15**

RS-274X parameters 15
  AD - Aperture Definition 16
  AM - Aperture Macro 19
  AS - Axis Select 25
  FS - Format Statement 26
  IF - Include File 28
  IJ - Image Justify 29
  IN - Image Name 30
  IO - Image Offset 31
  IP - Image Polarity 32
  IR - Image Rotate 33
  KO - Knockout 34
  LN - Layer Name 36
  LP - Layer Polarity 37
  MI - Mirror Image 38
  MO - Mode 39
  OF - Offset 40
  PF - Plot Film 41

SF - Scale Factor ... 42
SR - Step and Repeat ... 43

STANDARD RS-274D CODES ... 45

D Codes ... 45
Table 4 D Codes ... 45
G Codes ... 46
Table 5 G Codes ... 46
Linear Interpolation (G01, G10, G11, G12) ... 47
Circular Interpolation (G02, G03, G74, G75) ... 47
Multiquadrant (360°) Circular Interpolation (G74, G75) ... 48
Polygon Area Fill (G36, G37) ... 49
M Codes ... 50

GLOSSARY ... 51

INDEX ... 53

Introduction

Gerber data is a simple, generic means of transferring printed circuit board information to a wide variety of devices that convert the electronic PCB data to artwork produced by a photoplotter. Virtually every PCB CAD system generates Gerber data because all photoplotters read it. It is a software structure consisting of X,Y coordinates supplemented by commands that define where the PCB image starts, what shape it will take, and where it ends. In addition to the coordinates, Gerber data contains aperture information, which defines the shapes and sizes of lines, holes, and other features.

Gerber Format, which is the format in which Gerber data is expressed, actually is a family of data formats that are subsets of EIA Standard RS-274D. *Extended Gerber Format*, which is also called *RS-274X*, provides enhancements that handle polygon fill codes, positive/negative image compositing, and custom apertures, and other features. RS-274X also encapsulates the aperture list in the header of the Gerber data file and therefore allows files to pass from one system to another without the need to re-input the aperture table. RS-274X produces a variety of Gerber data called *X data*.

RS-274X is a *superset* of the EIA Standard RS-274D format. RS-274X supports some of the parameter data codes (G codes) and aperture codes (D codes) contained in RS-274D, as well as codes referred to as *mass parameters*. Mass parameters are plot parameters that define characteristics that can affect an entire plot, or only specific parts of the plot, called *layers.* Mass parameters extend the capabilities of Gerber Format. Their presence makes the Gerber data *X data.*

RS-274X is maintained by Gerber Systems Corporation (GS), a leading supplier of CAD/CAM systems, large-area plotting systems, and precision cutting systems since 1965.

Who should use this Guide?

In order to use this Guide, you should have a fundamental understanding of PCB fabrication or PCB design and laser plotting concepts. This Guide is intended for use by:

- PCB designers preparing data for conversion to RS-274X
- PCB fabricators creating or using Gerber data files
- Developers of software applications using RS-274X data

How to use this Guide

You will find the following sections in this Guide:

Rules and Guidelines explain file content and structure and outlines rules and guidelines for creating a correct RS-274X file. It also contains a sample file.

Reference defines use and constraints on use of every RS-274X parameter and data code currently supported. Parameters and data codes are described separately. Both are presented in alphanumerical order.

You will also find a **Glossary** and **Index** at the end of the Guide.

Related documentation

This Guide assumes you are familiar with Electronic Industries Association EIA Standard RS-274D. You can obtain a copy of this standard from the Electronic Industries Association, Engineering Department, 2001 Eye Street NW, Washington, DC 20006 USA.

Where to go for help

Should you need assistance, contact the Baraco Gerber Systems Corporation Technical Assistance Center by telephone 8 a.m. to 5 p.m. (eastern time) at (860) 291-7016, by fax at (860) 291-7021.

Rules and Guidelines

This section provides background information, describes organization, and presents guidelines for use of RS-274X. For detailed descriptions of use of individual codes and parameters, see *Reference*, page 15.

File structure

An *RS-274X plot file* is a file consisting of RS-274X parameters and standard RS-274D codes which, when correctly interpreted, result in an image that may be displayed or plotted.

Data blocks

The file is composed of a number of *data blocks* containing parameters and codes. Each data block is delimited by an end-of-block character, typically an asterisk (*).

Each data block may contain one or more parameters or codes. For example:

```
X0Y0D02*
X50000Y0D01*
```

Layers

One or more data blocks may be grouped into a *layer* of information that describes part of a graphic image. In RS-274X context, a layer is a *named* ***information*** *component of the image* composed of one or more data blocks. Each layer may have characteristics, such as name, polarity, and interpolation mode, that differ from other layers of information . In addition, an individual layer may be "knocked out "of the surrounding graphic image, and may be repeated and/or rotated individually.

Note: A layer must not be confused with a PCB layer. A PCB layer has a physical definition and might be compared to a physical plane. An RS-274X layer is simply a group of data blocks that may be manipulated collectively and separately from other layers.

Data types

An RS-274X file may contain the following kinds of data appearing in the following general order:

1. **RS-274X Parameters**

   RS-274X parameters are also called *mass parameters* or *extended Gerber format*. The inclusion of these parameters in the file makes the plot file RS-274X, or *X data*, instead of standard RS-274D.

2. **Standard RS-274D Codes**

   Standard RS-274D codes were once called *word address format*. They consist of:

   - one-character **function codes** such as G codes, D codes, M codes, etc. Function codes were the *words* of the old terminology. They describe how coordinate data associated with them should be interpreted (such as linear or circular interpolation), how the imaging device should move (light source on or off), and more.
   - **coordinate data** define points to which the imaging device must move. The coordinate data represented the *address* of the old terminology. X,Y coordinate data describe linear positions. I, J coordinates define arcs.

RS-274X parameters

RS-274X parameters define characteristics that apply to an entire plot or to a single layer, depending on the parameter's position in the file and whether it generates a new information layer in the file (as, for example, layer-specific parameters do). RS-274X parameters consist of two alpha characters followed by one or more optional modifiers.

RS-274X parameters are delimited by a *parameter delimiter*, typically a percent (%) sign. Because parameters are also contained in a data block, they are also delimited by an end-of-block character. For example,

```
%FSLAX23Y23*%
```

This parameter is a Format Statement (FS) describing how the coordinate data in the file should be interpreted (in this case, 4.2 format for both X and Y coordinates). It is delimited by an end-of block character (*) as well as by a parameter delimiter (%).

RS-274X parameters may be grouped according to the scope of their function in the file. The groups should appear in the file in the following order:

1. **Directive parameters** control overall file processing.
2. **Image parameters** supply information about an entire image.
3. **Aperture parameters** describe the shape of lines and components throughout the file.
4. **Layer-specific parameters** describe processing of one or more data layers.
5. **Miscellaneous parameters** provide capabilities that do not fall into the above groups.

RS-274X parameters are generally placed at the beginning of the file in the order shown above. Certain parameters, such as the layer-specific parameters, may be embedded within the file.

The next sections describe each parameter type.

Directive parameters

Directive parameters control overall file processing. They include:

AS	Axis Select
FS	Format Statement
MI	Mirror Image
MO	Mode of units
OF	Offset
SF	Scale Factor

As a general rule, directive parameters should be placed at the beginning of the file. For simplicity sake, you should use each directive parameter only once in a file, although it is not illegal to use a directive parameter more than once. Each directive parameter controls processing until another like it is encountered.

When used more than once in a file, subsequent directive parameters may be embedded anywhere in the standard RS-274D code data or grouped with other layer-specific parameters.

Directive parameters do not generate a new layer.

Image parameters

Image parameters supply information about the entire (composite) image. Image parameters include:

IJ	Image Justify
IN	Image Name
IO	Image Offset
IP	Image Polarity
IR	Image Rotation
PF	Plotter Film

Image parameters should be used only once in a file and should be placed at the beginning of the file. If an image parameter occurs more than once in a file, the last one encountered will be the operative parameter.

Aperture parameters

Vector plotters control the width and shape of features by projecting light through a series of openings, or apertures, in a rotating wheel. Each position on the wheel is identified by a unique D code. When the D code appears in the data, the wheel rotates to the referenced position for exposure.

Unlike a vector device, a raster device has no apertures and therefore requires a description of the aperture geometry to create the required lines and other features. The aperture parameters provide the description.

The aperture parameters include:

AD	Aperture Description
AM	Aperture Macro

In general, aperture parameters apply to an entire file. An exception is an embedded AD parameter, which will generate a new layer if it redefines a D code previously used in the image data.

Note: Generating a new layer may result in unanticipated results because it causes certain RS-274D values (such as interpolation mode) to be reset.

The AM parameter describes a special aperture by using the following set of predefined aperture shapes to describe an aperture:

Circle
Line (vector)
Line (center)
Line (lower left)
Outline
Polygon
Moiré
Thermal

See the AM command description, page 19, for more information.

Layer-specific parameters

Layer-specific parameters supply information for the processing of one or more *information* layers (not to be confused with board layers). They may be used more than once in a file. Layer-specific parameters always generate a new layer and should be placed at the beginning of the new layer. If not repeated for a given layer, the previous layer-specific parameters remain in effect.

The layer-specific parameters include:

KO	Knockout
LN	Layer Name
LP	Layer Polarity
SR	Step and Repeat

Note: Generating a new layer may result in unanticipated results because it causes certain RS-274D values (such as interpolation mode) to be reset.

Miscellaneous parameters

There is a single miscellaneous parameter:

IF	Include File

The IF parameter is used to include (nest) external files in a file.

Standard RS-274D Codes and Coordinate Data

Standard RS-274D codes (D codes, G codes, M codes, etc.) specify how the coordinate data should be manipulated. Each code applies to coordinate data located in the same data block as the code (that is, between EOB characters). It also applies to coordinate data following it until another code of the same type is encountered, or until a new layer is generated. This continuing action is referred to as *modal*.

For example, G02 specifies clockwise, single-quadrant circular interpolation and is modal. All coordinate data following it will be considered clockwise arc data until another interpolation code is encountered, or until a new layer is generated. When a new layer is generated, interpolation will be reset to linear (G01).

Like parameters, standard RS-274D codes may be grouped according to function. They generally appear in the file in the following sequence:

1. **N codes** (sequence numbers) are similar to line numbers and may be assigned to data blocks to simplify organization. Sequence numbers may be 0 to 99999. N codes are not necessary.
2. **G codes** (general functions) specify how to interpolate and move to the coordinate locations following the code until changed or until a new layer is generated (modal).
3. **D codes** (plot functions) select and control tools, specify line type, etc.
4. **M codes** (miscellaneous functions) perform a variety of functions such as program stop and origin specification.

Standard RS-274D codes are described in detail starting on page 45.

Coordinate Data

Coordinate data includes:

- **X,Y data** define linear positions along the X and Y axes.
- **I,J data** define arcs.

For example,

```
X200Y200D02*
```

This data block directs the plotter to move in a positive direction to coordinate location 0.2,0.2 (assuming leading zeroes are omitted) with the light source off (tool up). Additional X,Y coordinate data positions following this code will also cause motion with the light source off until a different code is encountered.

Absolute versus relative data

Depending on the preceding FS parameter in the file, coordinate data may be defined either relative to the plot origin (that is, as absolute values) or relative to the last coordinate position (that is, as incremental data).

Numerical precision

Coordinate data may be expressed in inches or millimeters to ±6.6 decimal places (that is, up to six integer digits and six fractional digits). Unless preceded by a "-", the direction is assumed to be "+".

Axis assignment

The coordinate axes may be assigned to any physical plotter axes using the AS parameter, but typically the A plotter axis is assigned to X and the B axis to Y.

General File Preparation Guidelines

Follow these guidelines when preparing RS-274X data:

- Use data blocks in a way that organizes the file visually for easy reading.
- Enter all codes and parameters in upper case.
- Use file names that comply with the system file naming conventions. DOS and therefore Windows 3.1 files names are limited to eight characters. UNIX systems are case-sensitive.
- End every data block with an end-of-block character, typically *. For example,

```
X0000Y5000D02*
```

- Do not break a line within a block.
- End every file with an end-of-program code (M00 or M02).

RS-274X Parameter Guidelines

- Begin and end RS-274X parameter data with a parameter delimiter, typically %. The parameter delimiter must immediately follow the end-of-block without intervening spaces. For example,

```
%ASAXBY*%
```

- Parameters may be entered singly or grouped between delimiters, up to a maximum of 4096 characters between delimiters. A maximum of 80 characters between delimiters is recommended. Always consider readability. For example,

```
%SFA1.0B1.0*ASAXBY*%
```

- Line breaks are permitted between parameters to improve readability. For example,

```
%SFA1.0B1.0*
ASAXBY*%
```

- Use an explicit decimal point with all numerical values associated with a parameter. If the decimal point is omitted, an integer value is assumed.
- Express numerical values in the units defined by the MO code in the file (inches or millimeters).

- In general, enter RS-274X parameters at the beginning of the file (or at the beginning of the layer for layer-specific parameters). Enter them in the order shown in Table 1.

Note: When RS-274X parameters are embedded in the RS-274D data, all data preceding the parameter will be processed before the data blocks affected by the embedded parameters are interpreted.

Table 1 RS-274X Parameter Order of Entry

Parameter Required	Parameter Optional	Function	Comments	Default
	AS	Axis select	Single use recommended. When used more than once, enter these parameters at the beginning of a layer. These codes do not generate a new layer.	A=X, B=Y
FS		Format statement		
	MI	Mirror image		No mirror
	MO	Mode (inch or millimeter units)		Inch
	OF	Offset		A=0, B=0
	SF	Scale Factor		A=1.0, B=1.0
	IJ	Image Justify	Use only once at the beginning of the file.	No justification
IN		Image Name		
	IO	Image Offset		A=0, B=0
	IP	Image Polarity		Positive
	IR	Image Rotation		0
	PF	Plotter Film		
AD		Aperture Description	May be used singly or may be layer-specific. Enter these parameters at the beginning of the file or layer.	
	AM	Aperture Macro		
	LN	Layer Name		
	LP	Layer Polarity		Positive
	KO	Knockout		Off
	SR	Step and Repeat		A=1, B=1
	RO	Rotate		No rotation

RS-274D Data Guidelines

Follow these guidelines when preparing RS-274D data:

- Enter functions codes and coordinate data following the RS-274X parameters.
- Function codes apply to coordinate data in the same block as well as to subsequent coordinate data. They do not affect coordinate data preceding the block in which they occur.
- Enter function codes in the file in the order shown in Table 2.

Table 2 RS-274D Code Order of Entry

Code	Function	Comments
N	Sequence number	Optional
G	General functions: linear interpolation, circular interpolation, polygon area fill, etc.	Once encountered, remains in effect until countermanded.
D	Aperture or tool assignment; line/flash control	Once encountered, remains in effect until countermanded.
M	Miscellaneous function: program stop or end.	Every file must end with M00 or M02.

Sample Files

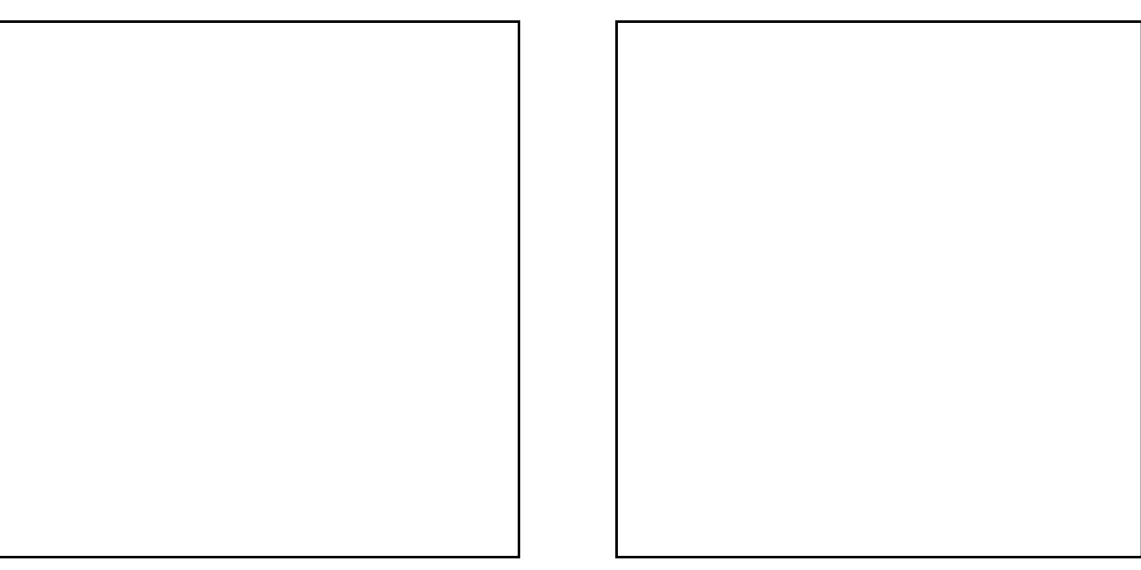
The examples on these pages illustrate the use of both mass parameters and standard RS-274D codes.

Example 1

Example 1 illustrates a single layer image.

Code	Comment
***G04 EXAMPLE 1: 2 BOXES**	
%FSLAX23Y23*%	Format statement - leading zeroes omitted, absolute coordinates, X2.3, Y2.3.
%MOIN*%	Set units to inches.
%OFA0B0*%	No offset
%SFA1.0B1.0*%	Scale factor is A1, B1
%ADD10C,0.010*%	Define aperture D-code 10 - 10 mil circle
%LNBOXES*%	Name layer “BOXES”.
G54D10	
X0Y0D02*X5000Y0D01*	
X5000Y5000D01*X0Y5000D01*X0Y0D01*	RS-274D data
X6000Y0*X11000Y0D01*	

X11000Y5000D01*X6000Y5000D01*
X6000Y0D01*D02*
M02* End of data



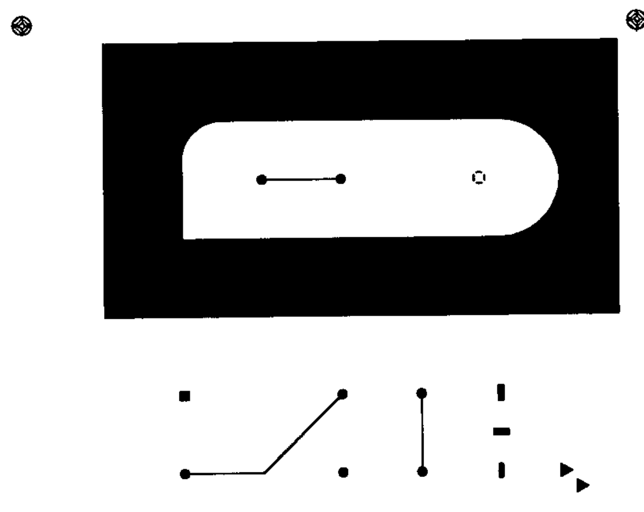
Example 2

Example 2 illustrates RS-274-X data.

%ASAXBY*	Axis Select, A=X, B=Y
FSLAX23Y23*	Format Statement, Leading zeros omitted, absolute data, 2 integer digits and 3 fractional digits
MIA0B0*	Mirror about the specified axis; 0=no, 1=yes
MOIN*	Mode inches
OFA0B0*	Offset 0
SFA1.0B1.0*%	Scale Factor
%IJALBL*	Image justify
INXTEST*	Image name
IOA0B0*	Image offset
IPPOS*	Image Polarity
IR0*%	Image Rotation
G04 Define Apertures*	Comment
%AMTARGET125*	Aperture Macro
6,0,0,0.125,.01,0.01,3,0.003,0.150,0*%	Moiré Description
%AMTHERMAL80*	Aperture Macro
7,0,0,0.080,0.055,0.0125,45*%	Thermal Description
%ADD10C,0.01*	Aperture Description, D10 is a circular aperture with 0.01” diameter
ADD11C,0.06*	Aperture Description, D11 is a circular aperture with 0.06” diameter
ADD12R,0.06X0.06*	Aperture Description, D12 is a rectangular aperture, 0.06” X 0.06”
ADD13R,0.04X0.100*	Aperture Description, D13 is a rectangular aperture, 0.04” X 0.100”
ADD14R,0.100X0.04*	Aperture Description, D14 is a rectangular aperture, 0.100” X 0.04”
ADD15O,0.04X0.100*	Aperture Description, D15 is a obround aperture, 0.04” X 0.100”
ADD16P,0.100X3*	Aperture Description, D16 is a 3 sided polygon 0.100” overall size
ADD17P,0.100X3*	Aperture Description, D17 is a 3 sided polygon 0.100” overall size
ADD18TARGET125*	Aperture Description, D18 is a special aperture called "TARGET”

ADD19THERMAL80*%	Aperture Description, D19 is a special aperture called "THERMAL”
%LNXTEST1*	Layer Name XTEST1
LPD*	Layer Polarity Dark
SRX1Y1I0J0*%	Step and Repeat set to 1 X 1 (Not Required)
G54D10*	Aperture select
G01X0Y250D02*	Linear move with light off
X0Y0D01*	Linear move with light on
X250Y0D01*	Linear move with light on
X1000Y1000D02*	Linear move with light off
X1500D01*	Linear move with light on
X2000Y1500*	Notice since D01 is modal it does not need to be repeated
X2500D02*	Notice since the X & Y commands are modal, Y is not repeated
Y1000D01*	Here, X is not repeated and uses its previous value of 2.500”
D02*	Light off no move
G54D11*	New aperture selected
G55X1000Y1000D03*	G55 prepares for flash It is not necessary. D03 is the flash command.
X2000D03*	Y value does not change
X2500D03*	This method reduces the size of the file
Y1500D03*	Here, X does not change from previous value
X2000D03*	Flash
G54D12*	New aperture select
X1000Y1500D03*	Move to (1.0, 1.5) and flash
G54D13*	New aperture select
X3000Y1500D03*	Move and flash
G54D14*	New aperture select
Y1250D03*	Move and flash
G54D15*	New aperture select
Y1000D03*	Move and flash
G54D10*	New aperture select
G01X3750Y1000D02*	Linear move, light off. Start point of the following arc command
G75*	Sets the mode to 360 degree circular interpolation
G03X3750Y1000I250J0D01*	Move from start point above to end point drawing a complete circle
G54D16*	New aperture select
G55X3400Y1000D03*	Flash
G54D17*	New aperture select
G55X3500Y900D03*	Flash
G54D10*	New aperture select
G36*	Start Polygon fill
G01X500Y2000D02*	
Y3750D01*	
X3750*	
Y2000*	
X500*	
X500Y2000D02*	
G37*	End Polygon fill
G54D18*	New aperture select
G55X0Y3875D03*	Flash
X3875Y3875D03*	Flash
%LNXTEST2*	Layer Name
LPC*%	Layer Polarity clear
G36*	Start Polygon fill
G01X1000Y2500D02*	
Y3000D01*	
G74*	Quadrant arc
G02X1250Y3250I250J0D01*	Clockwise arc move with radius .25”
G01X3000*	Complete 90 degree arc
G75*	360 degree arc mode

G02X3000Y2500I0J-375D01*	Clockwise arc move with radius .375”
G01X1000*	Linear move light on
X1000Y2500D02*	Linear move light off
G37*	End Polygon fill
%LNXTEST3*	Layer Name
LPD*%	Layer Polarity Dark
G54D10*	New aperture select
X1500Y2875D02*	
X2000D01*	
D02*	
G54D11*	
X1500Y2875D03*	
X2000D03*	
G54D19*	New aperture select
X2875Y2875D03*	
M02*	End of file



Reference

RS-274X parameters

This section describes every RS-274X parameter supported at time of publication. They are arranged in alphabetical order. Standard RS-274D code descriptions begin on page 45.

Each parameter description illustrates the parameter data block format, explains each parameter modifier, lists restrictions, and gives an example.

The data block format illustration uses the following notation conventions:

%Parameter code<required modifiers>[optional modifiers]*%

where:

Parameter code	is the 2-character code (AD, AM, FS, etc.)
<required modifiers>	must be entered to complete definition
[optional modifiers]	may be required depending on the required modifiers

The AD parameter is used to describe apertures (D codes) used in the RS-274X file. All apertures used in an RS-274X file must be described in terms of shape and size for the file to be interpreted correctly. The AD parameter must precede use of the associated aperture D-code. A definition remains in effect until redefined.

Two kinds of apertures may be used in an RS-274X file: *standard* apertures and *special* apertures.

Standard apertures

The AD parameter identifies standard apertures by D-code number and describes them in terms of shape (circular, rectangular, obround, or polygonal) and size (diameter if round, height and width if rectangular or obround, outside dimension and number of sides if polygonal). Apertures may be solid or open (that is, with a hole) and are always centered.

Special apertures

The AD parameter is also used to assign a D-code to a file containing an aperture description. The aperture description file may be a .mac file created by the AM (Aperture Macro) parameter or a .des file created by an Aperture Editor such as the Gerber GPC Aperture Editor. See the AM parameter description for further information on aperture macros.

AD parameter syntax rules

- Like other mass parameters, begin and end each parameter block with a parameter delimiter (typically %).
- Within the AD parameter block, separate each modifier by an X.
- Dimensions must be positive and will be rounded to the resolution of the output device.
- The various plotters and output devices may permit different D-code ranges, but the range must not exceed 10 to 999.

Data Block Format

```
%ADD<D-code number><aperture type>,<modifier>[X<modifer>]*%
```

where:

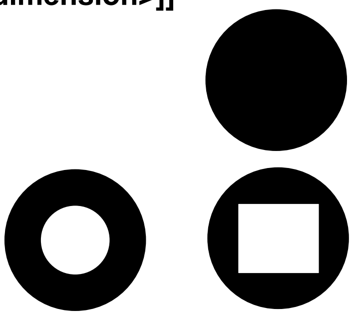
ADD	the AD parameter and D (for D-code)
<D-code number>	the D-code number being defined (10 - 999)
<aperture type>,<modifier>[X<modifier>]	the aperture descriptions. **<aperture type>** may be one of the following: • a standard aperture (C, R, O, P, or T) with modifiers (see below). Modifiers depend on aperture type. Use **X** to separate each modifier. All dimensions are positive and will be rounded to the resolution of the output device.

- a file name containing the aperture description (.des file)
- an aperture macro name previously defined by the AM parameter (.mac file)

Note: Be sure to use the units (inches or millimeters) specified by the MO parameter for all modifiers.

Standard apertures:

C, **<outside diameter>[X<X-axis hole dimension >[X<Y-axis hole dimension>]]**

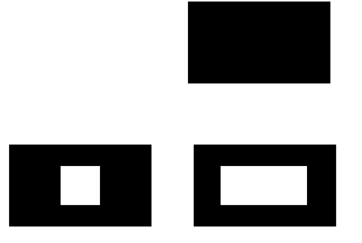


Circle. To define a solid aperture, enter only the diameter. To define a hole, enter one dimension for a round hole, two for a rectangle. The hole must fit within the aperture. For a square hole, X must equal Y. Both aperture and hole will be centered. For example,

%ADD10C,.05X0.025*%

D-code 10 is a .05 circle with a .025 round hole in the center.

R, **<X-axis dimension>X<Y-axis dimension>[X<X-axis hole dimension>X<Y-axis hole dimension>]**

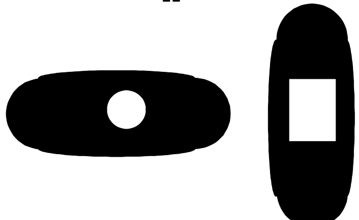


Rectangle or square. May be solid or open. If the X axis dimension equals the Y dimension, the aperture will be square. To define a solid aperture, enter only the X and Y dimensions; omit the hole dimensions. To define a hole, enter one dimension for a round hole, two for a rectangle. The hole must fit within the aperture. Both rectangle and hole will be centered. For example,

%ADD22R,0.020X0.040*%

D-code 22 is a .02 x .04 solid rectangle.

O, **<X-axis dimension>X<Y-axis dimension>[X<X-axis hole dimension>[X<Y-axis hole dimension>]]**

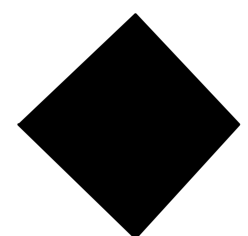


Obround (oval). May be solid or open. If the X dimension is larger than Y, the shape will be horizontal. If the X dimension is smaller than Y, the shape will be vertical. To define a solid aperture, enter only the X and Y dimensions; omit the hole dimensions. To define a hole, enter one hole dimension for a round hole, two for a rectangular or square hole. If open, the hole must fit within the aperture. For example,

%ADD22O,0.020X0.04X0.005X0.010*%

D-code 22 is a vertical obround .02 wide x .04 high with a .05 x .01 rectangular hole.

P,	**<outside dimension>X<number of sides>[X<degrees of rotation>[X<X-axis hole dimension>X<Y-axis hole dimension>]]**	**Regular polygon.** May be solid or open. To define a solid aperture, enter only the outside dimension and number of sides (3 to 12). The first point is located on the X axis. May be rotated ±360° from the X-axis. If open, the hole must fit within the outside dimension. *Note: If you use the hole dimension modifiers, you must enter a rotation (even if it is 0).* For example,



%ADD17Diamond,.030X4X0.0*%

D-code 17 is a polygon within an outside dimension of .03, 4 sides, with no center hole.

Examples

%ADD10C,.025*%	Define D-code 10: 25 mil round
%ADD22R,.050X.050X.027*%	Define D-code 22: 50 mil square with 27 mil round hole
%ADD57O,.030X.040X.015*%	Define D-code 57: obround 30 x 40 mil with 15 mil round hole
%ADD30P,.016X6*%	Define D-code 30: polygon (hexagon), 16 mil outside dimension with 6 sides
%ADD15CIRC*%	Define D-code 15: a special aperture described by aperture macro CIRC defined previously by an aperture macro

The AM parameter is used to define named apertures (sometimes called *special apertures*) in *aperture macro* format consisting of building blocks called *primitives*. The named aperture macros may be used in AD parameter descriptions just like the standard apertures (that is, circle, rectangle, obround, polygon, and thermal). Every non-standard aperture must be described before the D-code associated with it occurs in the file.

Special apertures offer two advantages over standard apertures:

- They allow multiple shapes called primitives to be combined in a single aperture, which permits creation of unusual or complicated apertures.
- They need not be centered.
- Aperture macro modifiers may be variable. Variable modifiers are supplied by the AD parameter that references the aperture macro.
- An aperture macro variable may be a numerical function of another macro variable (+, -, etc.).

Aperture macro contents

An aperture macro contains the following elements:

- aperture macro name
- one or more of the seven aperture primitives, each identified by a primitive number (see Table 3 below for a description of the primitives)
- primitive modifiers that describe the primitive in terms of exposure, position, dimensions, etc.
- variable primitive modifiers to be supplied by the AD parameter
- optional embedded comment blocks
- numerical operators

AM parameter syntax rules

- Like other mass parameters, begin and end each parameter block with a parameter delimiter (typically %).
- Within the AM parameter block, separate each primitive and modifier group by an end-of-block character (typically *).
- Within each primitive group, separate modifiers by commas.
- Modifiers may be absolute values, such as 0, 1, 2, or 9.05, or they may be variable modifiers to be supplied by the AD parameter when it refers to the aperture macro.
- Identify variable modifiers to be supplied by the AD parameter as *$n* where *n* indicates the order in which the modifier is expected in the AD parameter. $1

would be the first variable modifier expected in the AD parameter, $2 the second, and so on, numbering sequentially from left to right. If an absolute value is entered instead of a variable, the variables shift right. For example, if an absolute value is entered for the first variable, the next variable becomes $1 even though it is the second modifier of the primitive.

- The interpretation of each modifier differs for each primitive. See Table 3 on the next page for a full explanation of aperture macro primitives and modifiers.
- Do not begin a variable primitive modifier with a minus sign (for example, -$1). To indicate negative, precede the variable with 0 (for example, **0-$1**).
- Start optional comment strings with a leading 0 (for example, ***0 THIS IS A COMMENT*).**
- Position and dimensions are expressed in the units specified by the MO parameter. Decimal points are permitted.
- Use only the following numerical operators with variable modifiers:

Operator	Function
+	add
-	subtract
/	divide
x	multiply
=	equate
n	numerical factor

- Make sure the aperture macro file name matches the aperture macro name and that it has a .mac extension.

Data Block Format

%AM<aperture macro name>*<primitive number>,<modifier$1>,<modifier$2>,[<...>]*[<primitive number>[<modifiers>]]*...*%

where:

AM	the AM parameter
<aperture macro name>*	the name to be used in the AD parameter
<primitive number>,<modifier$1>,<modifier$2>,<modifier$3>,...*	the primitive number with modifiers. The primitive number identifies the geometry (outline, polygon, etc.). The modifiers differ with the various primitive numbers. Use either actual values (for example, 0.050 for a width) or a variable placeholder (for example, $1 for exposure on/off).

Table 3 Aperture macro primitives

Primitive number	Description	Variable Modifiers	Description
1	**Circle**	$1	Exposure on/off 0 = OFF (laser off, no exposure) 1 = ON (laser on, image exposed) 2 = reverse current exposure state
		$2	Diameter
		$3	X center position
		$4	Y center position
2 or **20**	**Line (vector)**: a line defined by width, and beginning and end points. The line ends are rectangular.	$1	Exposure on/off 0 = OFF (laser off, no exposure) 1 = ON (laser on, image exposed) 2 = reverse current exposure state
		$2	Line width
		$3	X start point
		$4	Y start point
		$5	X end point
		$6	Y end point
		$7	Rotation in degrees (+ = counterclockwise, - = clockwise)
21	**Line (center)**: a centered rectangle defined by width, height, and center point. The end points are rectangular.	$1	Exposure on/off 0 = OFF (laser off, no exposure) 1 = ON (laser on, image exposed) 2 = reverse current exposure state
		$2	Rectangle width
		$3	Rectangle height
		$4	X center point
		$5	Y center point
		$6	Rotation in degrees (+ = counterclockwise, - = clockwise)

22	**Line (lower left)**: a rectangle defined by width, height, and the lower left point. The end points are rectangular. $4,$5 $3 $2	$1	Exposure on/off 0 = OFF (laser off, no exposure) 1 = ON (laser on, image exposed) 2 = reverse current exposure state
		$2	Width
		$3	Height
		$4	X lower left point
		$5	Y lower left point
		$6	Rotation in degrees (+ = counterclockwise, - = clockwise)
3	End of file	none	Must be used to end .des files.
4	**Outline**: an open or closed shape defined by a start point, *n* additional points (up to 50), and the X,Y coordinates that define them. For a closed shape, the first and last points must be identical. $3,$4 $7,$8 $5,$6	$1	Exposure on/off 0 = OFF (laser off, no exposure) 1 = ON (laser on, image exposed) 2 = reverse current exposure state
		$2	*n*, the number of points in the outline
		$3	X start point
		$4	Y start point
		$5	X point #1
		$6	Y point #1
		$7	X point #2
		$8, etc.	Y point #2. Continue as needed.
		$9 or the last number used	Rotation in degrees (+ = counterclockwise, - = clockwise)
5	**Polygon**: a closed, symmetrical, centered shape defined by n vertices (3 to 10 inclusive), a center point, diameter, and rotation. $3, $4 First Vertex $5	$1	Exposure on/off 0 = OFF (laser off, no exposure) 1 = ON (laser on, image exposed) 2 = reverse current exposure state
		$2	number of vertices (integer)
		$3	X center point
		$4	Y center point
		$5	Diameter
		$6	Rotation in degrees (+ = counterclockwise, - = clockwise)

6	**Moiré**: a cross hair centered on *n* concentric circles defined by the center point, outside diameter, line thickness, and gap between circles.	$1	X center point
		$2	Y center point
		$3	Outside diameter
		$4	Circle line thickness
		$5	Gap between circles
		$6	number of circles
		$7	Cross hair thickness
		$8	Cross hair length
		$9	Rotation in degrees (+ = counterclockwise, - = clockwise)
7	**Thermal**: a cross hair centered on a circle defined by outside and inside diameter.	$1	X center point
		$2	Y center point
		$3	Outside diameter
		$4	Inside diameter
		$5	Cross hair thickness
		$6	Rotation in degrees (+ = counterclockwise, - = clockwise)

Example 1

%AMDONUT*1,1,$1,$2,$3*1,0,$4,$2,$3*% Define an aperture macro named DONUT consisting of two concentric circles:

1,1,$1,$2,$3 Circle (1), exposure on (1), diameter ($1), X center ($2), Y center ($3) all to be supplied by AD parameter

1,0,$4,$2,$3 Circle (1), exposure off (0), diameter ($4, different from first circle), X center and Y center ($2 and $3, same as first circle)

The AD parameter using this macro might look like the following:

%ADD32DONUT,0.100X0X0X0.080*% Define D-code 32 to be aperture macro **DONUT**. The diameter of the first circle will be 0.100. The center of both circles will be at 0,0. The diameter of the second circle will be 0.080.

$1 = 0.100
$2 = 0
$3 = 0
$4 = 0.080

Example 2

%AMDONUT*1,1,$1,$2,$3*$1=$2+0.030*1,0,$1-$4,$2,$3*%

Define an aperture macro named **DONUT** consisting of two concentric circles with diameter of the second circle defined as a function of the diameter and center point of the first:

1,1,$1,$2,$3 Circle (1), exposure on (1), diameter ($1), and center point X,Y ($2, $3) to be defined in the AD parameter

$1=$2+0.030 Define a variable to be used to calculate the diameter of second circle to be a function of the diameter and center point X coordinate of the first.

1,0,$1-$4,$2,$3 Circle (1), exposure off (0), diameter ($1-$4), and center point X,Y ($2, $3, same as the first circle).

The ADD parameter using this aperture macro might look like:

%ADD33DONUT,0.020X0X0X0.014*% Define D-code 33 to be aperture macro **DONUT**. The diameter of the first circle 0.020. The center of both circles will be 0,0. The diameter of the second circle will be ((0 + 0.030) - 0.014).

Example 3

%AMDONUT*1,1,0.100X0X0*1,0,0.080X0X0*%

Define an aperture macro named **DONUT** consisting of two concentric circles, using primitive modifiers.

%ADD32DONUT*% The resulting AD command only needs to reference the aperture macro name.

The AS parameter is used to assign any two data axes to the output device A or B axes.

Data Block Format

AS A<X or Y>B<X or Y>*

where:

A and B are output device axes

X and Y are data axes

Default

AXBY

Example

%ASAYBX*% Assign the X axis data to the output device B axis and the Y axis data to the output device A axis.

The FS parameter is used to define the format of the input coordinate data and to define the allowable N, G, D, and M code lengths. It should be the first RS-274X parameter in the file. It is recommended that only one be used in the file. It is usually the first parameter.

The FS parameter allows you to specify the following format characteristics:

- Number of integer and decimal places in coordinate data (coordinate format)
- Zero omission (leading or trailing zeroes omitted)
- Absolute or incremental coordinate notation
- Sequence number (N-code) length
- General function code (G codes) length
- Draft code (D code) length
- Miscellaneous code (M code) length

Note: Decimal point programming is not supported.

Coordinate format

Coordinate format specifies how many integer and how many decimal places to expect in the coordinate data. For example, 2.3 format specifies two integer and three decimal places. A maximum of six integer and six decimal places may be specified (999999.999999). Different formats may be defined for the X and Y axes.

Zero omission

Zero omission compresses data by omitting either leading or trailing zeroes from coordinate values. Any given string of digits may be interpreted very differently depending on the zero omission specification. Coordinate format also affects how zero omission is interpreted.

Leading zero omission eliminates all zeroes that precede non-zero digits but retains following zeroes. For example, with 2.3 coordinate format, *15* would be interpreted as *0.015*.

Note: Use leading zero omission for NO ZEROES OMITTED files.

Trailing zero omission eliminates all zeroes following non-zero digits but retains preceding zeroes. For example, with 2.3 coordinate format, *15* would be interpreted as *15.000*.

Absolute or incremental notation

Coordinate values may be expressed as either absolute distances from a fixed 0,0 point or as relative distances from the preceding coordinate position.

RS-274D code lengths

The FS parameter can be used to specify length limits for the following standard RS-274D codes:

N Sequence number
G General function
D Plot function
M Miscellaneous function

These codes are described starting on page 45.

Data Block Format

%FS<L or T><A or I>[Nn][Gn]<Xn><Yn>[Dn][Mn]

where:

FS	The FS parameter
<L or T>	Use **L** to omit leading zeroes. Use **T** to omit trailing zeroes.
<A or I>	Use **A** for absolute coordinate values. Use **I** for incremental coordinate values.
[Nn], [Gn], [Dn], and [Mn]	Enter the code and an integer length limit, for example, **N2** to specify two-digit sequence codes.
<Xn> and <Yn>	Enter **X** or **Y** and the number of integer and decimal places in the coordinate data for each axis, for example, **X23** for X-axis data with two integer and three decimal places (99.999). 6.6 is maximum. The X and Y axes may have different values.

Example

%FSLAX25Y25*%	Coordinate data will have leading zeros omitted (L) and be expressed as absolute (A) positions with two integer and five decimal places in both axes (X25Y25).

The IF parameter is used to identify an external file to be included in the RS-274X file. The files referenced by the IF parameter will be interpreted exactly as if they were included at the point of reference in the RS-274X file. Included files may also contain IF parameters, up to 10 levels of nesting.

The IF parameter is often used to include an external aperture file containing AD and AM parameters that describe the apertures used in the RS-274X file, sometimes referred to as an "external" aperture list. The IF parameter can also be used to include external data files, which allows you to merge multiple data files. Included files simplify the creation of standard plot sequences such as title blocks and coupons.

Data Block Format

%IF<filename.ext>*%

Examples

%IFCOUPON3.GBR*%	Include file COUPON3.GBR.
%IFCIRCL.mac*%	Include aperture macro file CIRCL.mac.
%IFAPT004.des*%	Include aperture description file APT004.des.

The IJ parameter is used to override the absolute data coordinates for final placement of the image on the output device. The image may be centered or may be placed at an absolute position relative to the lower left of the platen.

Note: When centered, the pixel coordinates for the platen reside in the first quadrant (+X and +Y). X and Y are positive numbers, greater than zero and less than the platen size.

When more than one IJ parameter appear in the data, the final entry encountered is the one used.

Data Block Format

%IJ[A<parameter >B<parameter>][<offset>]*%

where:

IJ	the Image Justify parameter	
A	the plotter A axis justification	
<parameter>	**L**	left or lower justification (default)
	C	center justification
B	The plotter B axis justification	
<parameter>	**L**	left or lower justification (default)
	C	center justification
<offset>	the starting position offset relative to 0,0	

Default

None

Examples

%IJ*%	Left justify in X and lower justify in Y.
%IJAC*%	Center justify in X, lower justify in Y.
%IJACB.100*%	Center justify in X, offset .1 units in Y.
%IJALB.10*%	Left justify in X, offset .1 units in Y.
%IJB.100*%	Same as previous example.
%IJA1B1*%	Offset image 1 unit in X and Y.

IN
Image Name

The IN parameter is used to assign a name of up to 77 alphanumeric characters to the entire image of the RS-274X file. Information layers may also be named; see the LN parameter.

Data Block Format

%IN<character string>*%

where:

<character string> up to 77 alphanumeric characters except the asterisk (*).

Examples

%INSOLDERMASK*%

%INPANEL_1*%

The IO parameter is used to offset an image from the 0,0 point. The offset is expressed as an increment in the units defined by the MO parameter along the output device A and B axis. The AS parameter is used to correlate data axes with output device axes. The offset may be different for each axis and may be entered for a single axis.

Data Block Format

%IOA<±n>B<±n>*%

where:

IO the Image Offset parameter

A<±n> the offset along the output device A axis

B<±n> the offset along the output device B axis

Default

%IOA0B0*%

Examples

%IOA1.0B1.5*% Offset the image 1.0 units along the A axis and 1.5 units along the B axis from 0,0.

%IOB5.0*% Offset the image 5.0 units along the B axis from 0,0.

IP
Image Polarity

The IP parameter is used to specify the positive or negative polarity of the entire file image. This *image polarity* differs from *layer polarity*, which is specified by the LP parameter and which applies only to one or more data layers of the entire image.

Data Block Format

%IP<POS or NEG>*%

where:

IP	the IP parameter
<POS or NEG>	Use **POS** for positive polarity, **NEG** for negative polarity.

Default

%IPPOS*%

Example

%IPNEG*%	Output the entire image with negative polarity.

The IR parameter is used to rotate the entire image counterclockwise in 90° increments about the 0,0 coordinate. All apertures follow the rotation. If you do not use the IR parameter, 0° rotation is assumed.

Data Block Format

%IR<90 or 180 or 270>

where:

IR	the IR parameter
<90 or 180 or 270>	Enter the desired value.

Default

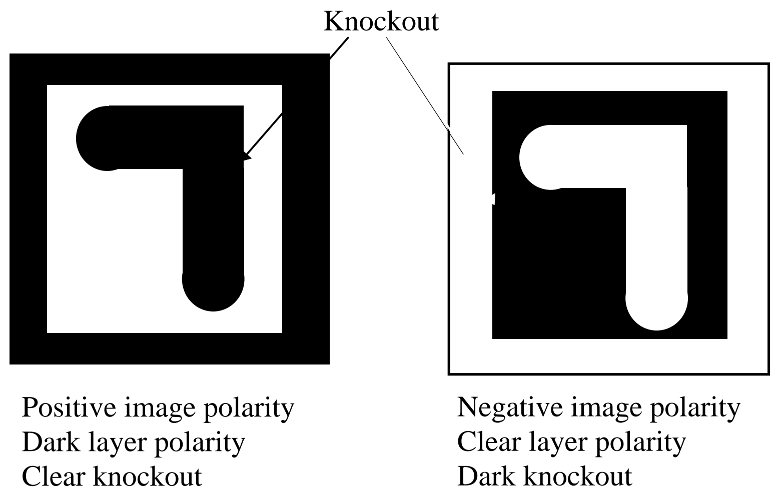
%IR0*%

Example

%IR90*%	Rotate the entire image 90° counterclockwise.

KO
Knockout

The KO parameter is used to specify that a rectangular region of the image will have the opposite polarity of the image or layer in which it occurs (either clear or dark), making it a "knockout" from the surrounding region.



Positive image polarity
Dark layer polarity
Clear knockout

Negative image polarity
Clear layer polarity
Dark knockout

KO is typically used to :

- set the initial background polarity of the final image
- set travel extent along the Y axis
- knock out a region around a component
- disable knockout on the current data layer

A knockout can be defined in one of two ways:

- lower left point and height and width
- border width around a component

When a knockout is defined around a component, the knockout is applied to all data following the KO parameter until the knockout is disabled. To disable a previously defined knockout, enter KO with no modifiers.

Data Block Format

%KO[C or D][X<coordinate>Y<coordinate>I<width>J<height>]or [K<border dimension>]

where:

KO — the KO parameter

C or D — Enter **C** for clear or **D** for dark. To create a knockout defined by the data extents, do not enter modifiers. To disable a previously enabled knockout, enter neither **C** nor **D**.

X<lower left X coordinate>Y<lower left Y coordinate>I<width>J<height>

Use this modifier to define the knockout by a lower left point, width, and height.

K<border dimension> Use this modifier to define the knockout as a border around a component. Express the dimension in units specified by the MO parameter.

Examples

%KODX0I0I20J26*% Create a dark knockout that extends from 0,0 to 20,26. This in effect sets the Y travel extent.

%KOCK.050*% Create a clear knockout .050 units around all sides of the data that follows the KO parameter until the knockout is disabled.

%KOD*% Make the region defined by the extents of the data following the command dark.

%KO*% Disable a previously enabled knockout.

LN Layer Name

The LN parameter is used to assign a name of up to 77 alphanumeric characters to the information layer that follows the parameter in the RS-274X file. Entire image files may also be named; see the IN parameter.

Data Block Format

%LN<character string>*%

where:

<character string> up to 77 alphanumeric characters except the asterisk (*).

Examples

%LNSOLDERMASK*%

%LNINTERNAL_VCC*%

The LP parameter is used to specify the positive or negative polarity of the information layer or layers following it. This *layer polarity* differs from *image polarity*, which is specified by the IP parameter and which applies to the entire image. The layer polarity applies to all data following the LP parameter until another LP parameter is encountered.

Data Block Format

%LP<C or D>*%

where:

LP the IP parameter

<C or D> Use **C** for clear polarity, **D** for dark polarity.

Default

%LPD*%

Example

%LPC*% Make all succeeding data clear.

The MI parameter is used to turn mirror imaging either on or off. When on, all A- and/or B-axis data following the parameter will be mirrored (that is, inverted or multiplied by -1) until another MI command is used. Notice that mirroring A-axis data flips the image about the B axis. Mirroring B-axis data flips the image about the A axis.

Note: MI does not mirror special apertures.

The AS parameter is used to correlate the X and Y axes with the output device A and B axes.

Data Block Format

%MI[A<0 or 1>][B<0 or 1>]*%

where:

MI the MI parameter

A<0 or 1> To invert A-axis data (flipping the image about the B-axis), enter A1. To disable, enter A0.

B<0 or 1> To invert B-axis data (flipping the image about the A-axis), enter B1. To disable, enter B0.

Default

%MIA0B0*%

Example

%MIA0B1*% Disable mirroring of A-axis data. Invert B-axis data, flipping the image about the A-axis.

The MO parameter specifies that dimension data should be interpreted as inches or millimeters. Integer and decimal place format is specified by the FS parameter. Inches are assumed if units are not specified.

Data Block Format

%MO<IN or MM>*%

where:

MO	the MO parameter
<IN or MM>	Enter **IN** to specify inches. Enter **MM** to specify millimeters.

Default

%MOIN*%

Example

%MOIN*%	Dimension data will be expressed in inches.

The OF parameter is used to offset the final image up to ±99999.99999 units from the imaging device 0,0 point. The data may be offset along the imaging device A or B axis, or both. Values used with the OF parameter are expressed in units specified by the MO parameter, are always absolute, and are used primarily with absolute coordinate data. Incremental coordinate data may be offset simply by moving the imaging device to the desired offset position before starting the plot. The FS parameter specifies whether the data is absolute or incremental.

If an embedded FS parameter changes the format from absolute to incremental, the OF parameter value is saved and reinstated another FS parameter returns the format to absolute.

Data Block Format

%OF[A<±offset value>][B±offset value>]

where:

OF	the OF parameter
A±n	Offset along the A axis. Use 5.5 format.
B±n	Offset along the B axis. Use 5.5 format.

Default

%OFA0B0*%

Example

%OFA1.0B1.0*%	Offsets the plot 1 unit from 0,0 along both the A and B axes.

PF
Plot Film

The PF parameter is used to indicate to the operator the film (or other media) to be used to image the data file.

Data Block Format

%PF<name>*%

where:

PF	the PF parameter
<name>	up to 20 alphanumeric characters; asterisk (*) is an illegal character

The SF parameter is used to specify a scale factor of from 0.0001 to 999.99999 for the output device A- and/or B-axis data. The factor may be different for each axis. All data following the parameter will be multiplied by the factor until another SF parameter is encountered. The AS parameter is used to correlate the X and Y data axes with the imaging device A and B axes.

Data Block Format

%SF[A<factor>][B<factor>]*%

where:

SF	the SF parameter
A<factor>	the A-axis data factor
B<factor>	the B-axis data factor

Default

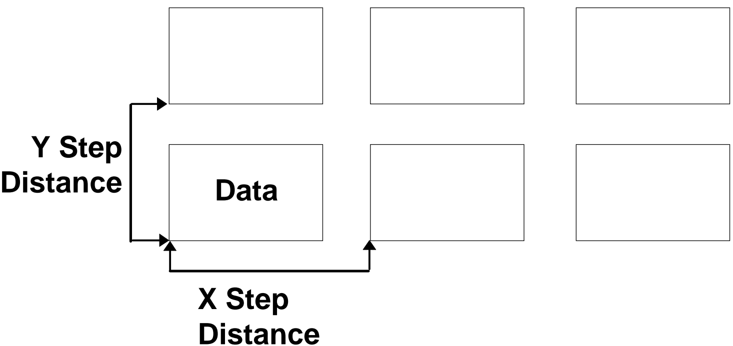
%A1B1*%

Example

%SFA.5B3*%	Multiply A-axis data by .5, B-axis data by 3.

The SR parameter is used to duplicate the data following the parameter a specific number of times (repeats) at a regular intervals (steps). The number of times the image is repeated and the space between repeats may be specified independently for X and Y data.

When entered without parameters, it is also used to disable a previous SR parameter.



Data Block Format

%SR[X<number of repeats>][Y<number of repeats>][I<X-axis step>][J<Y-axis step>]

where:

SR	the SR parameter
X<number of repeats>	the number of times the data will be repeated along the X-axis
Y<number of repeats>	the number of times the data will be repeated along the Y-axis
I<X-axis step distance>	the distance between the X-axis repeats
J<Y-axis step distance>	the distance between the Y-axis repeats

Default

%SRX1Y1I0J0*%

Examples

%SRX4I5.0J2*%	Repeat the image four times along the X axis with 5.0 units from one step to the next. The J modifier will be ignored because no Y repeats were specified.

%SRX2Y3I2.0J3*%	Repeat the image twice along the X axis and three times along the Y axis. X-axis repeats will be spaced 2.0 units apart. Y-axis repeats will be spaced three units apart.
%SR*%	Disable a previous SR parameter.

Standard RS-274D Codes

This section describes standard RS-274D codes (D codes, G codes, and M codes) that are applicable to raster output.

D Codes

D codes (draft codes) select apertures and determine whether the feature described should be imaged as a line or “flashed”. Table 4 lists supported D codes.

Table 4 D Codes

Code	Function	Comments
D01 (D1)	Draw line, exposure on	You cannot draw using an aperture defined by an aperture macro (AM parameter). These apertures can only be flashed.
D02 (D2)	Exposure off	
D03 (D3)	Flash aperture	D03 remains in effect until a new layer is encountered.
D10-D999	Select an aperture defined by an AD parameter.	

G Codes

G codes are general function codes. They specify how the coordinate data should be interpolated (linear or circular), turn the Polygon Area Fill feature on and off (see page 49 for more information on Polygon Area Fill), and can also be used to specify absolute or incremental format. Table 5 lists supported G codes.

Table 5 G Codes

Code	Function	Comments
G00	Move	Affected by Polygon Area Fill (see page 49.)
G01	Linear interpolation (1X scale)	See page 47.
G02	Clockwise circular interpolation	See page 47.
G03	Counterclockwise circular interpolation	
G04	Ignore data block	
G10	Linear interpolation (10X scale)	See page 47.
G11	Linear interpolation (0.1X scale)	
G12	Linear interpolation (0.01X scale)	
G36	Turn on Polygon Area Fill	See page 49.
G37	Turn off Polygon Area Fill	See page 49.
G54	Tool prepare	Usually precedes an aperture D-code
G70	Specify inches	See also MO parameter.
G71	Specify millimeters	See also MO parameter.
G74	Disable 360° circular interpolation (single quadrant)	See *Circular Interpolation*, page 47.
G75	Enable 360° circular interpolation (multiquadrant)	See *Circular Interpolation*, page 47.
G90	Specify absolute format	See also FS parameter.
G91	Specify incremental format	See also FS parameter.

Linear Interpolation (G01, G10, G11, G12)

Linear interpolation plots a straight line from the present position to the X,Y coordinate specified by the data block.

Data Block Format

G01 X±m.n Y±m.n Dnn

where:

G01 Specifies linear interpolation

X±m.n Y±m.n Defines the line end point

Dnn D-code (exposure on or off)

Circular Interpolation (G02, G03, G74, G75)

There are two types of circular interpolation: single quadrant (90°) and multiquadrant (360°). Single quadrant interpolation produces an arc. Multiquadrant interpolation can produce arcs that are larger than 90° and also circles.

Single Quadrant Circular Interpolation (G74)

G02 and G03 specify single quadrant (90°) circular interpolation; G74 disables it. Single quadrant circular interpolation plots an arc within one quadrant (90°). Single quadrant arcs must fit entirely within the quadrant in which they begin. A separate data block is required for each quadrant. A minimum of four data blocks is required to plot a circle.

Data Block Format

Gnn X±m.n Y±m.n Im.n Jm.n Dnn

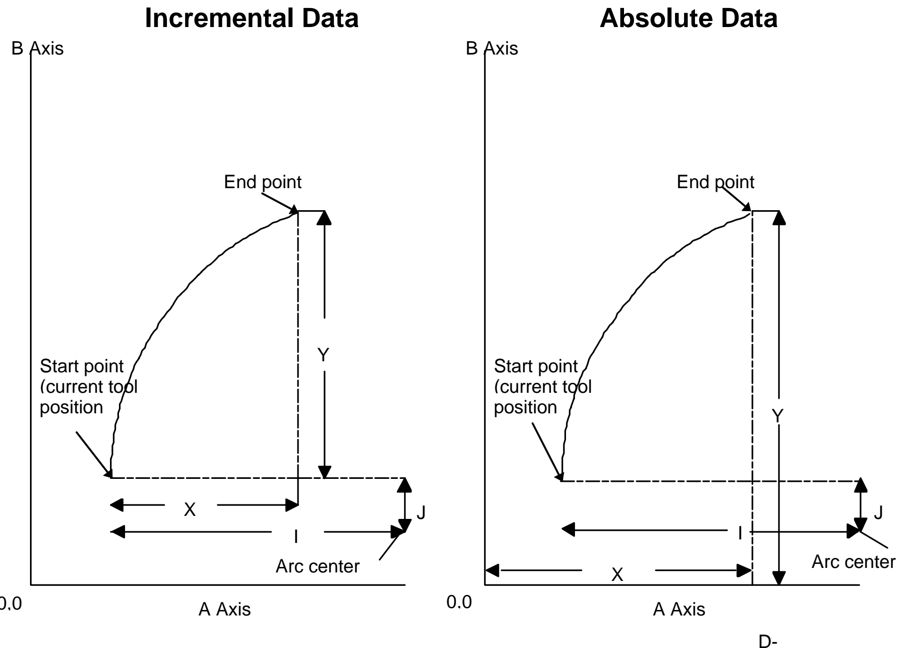
where:

Gnn G02 specifies clockwise circular interpolation
G03 specifies counterclockwise circular interpolation

X±m.n Y±m.n Defines the arc end point. These variables are in the format defined by the format statement (FS parameter). A sign is optional.

Im.n Jm.n Defines the incremental distance between the arc start point and the center measured parallel to the X and Y axes respectively. Notice that these numbers are unsigned values. The direction to the center is determined implicitly.

Dnn D-code (exposure on or off)



Multiquadrant (360°) Circular Interpolation (G74, G75)

A data block containing only G75 specifies 360° circular interpolation, which plots arcs in more than one quadrant using only one data block. Every block following a G75 code will be interpreted as 360° interpolation until a G74 is encountered. The I and J variables will be considered signed. If no sign is present, the circle will be in a positive direction from the start point.

A G74 code turns 360° multiquadrant circular interpolation off, reverting to single quadrant interpolation. To turn circular interpolation off and revert to linear interpolation, use G01.

Data Block Format

Gnn X±m.n Y±m.n ±Im.n ±Jm.n Dnn

where:

Gnn G74 turns off 360° circular interpolation
G75 turns on 360° circular interpolation

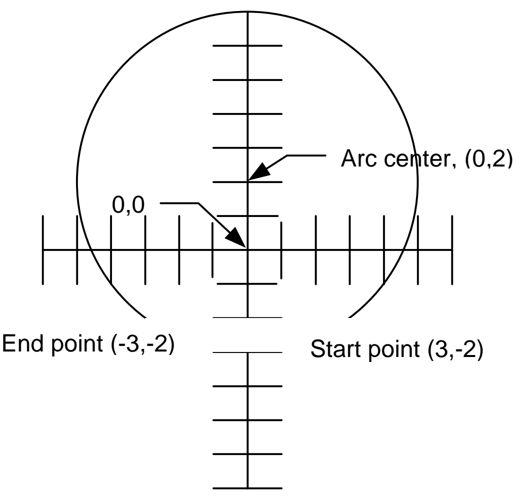
X±m.n Y±m.n Defines the arc end point

I±m.n J±m.n Defines the distance between the arc start point and the center measured parallel to the X and Y axes respectively. These variables are always incremental values in the format defined by the format statement (FS parameter). A sign is optional.

Dnn D-code (exposure on or off)

An example of multiquadrant interpolation is shown on the next page.

```
%FSLAX43Y43*%
G75*
G01X3000Y-2000D02*
G03X-3000Y-2000I-3000J4000D01
G01*
```

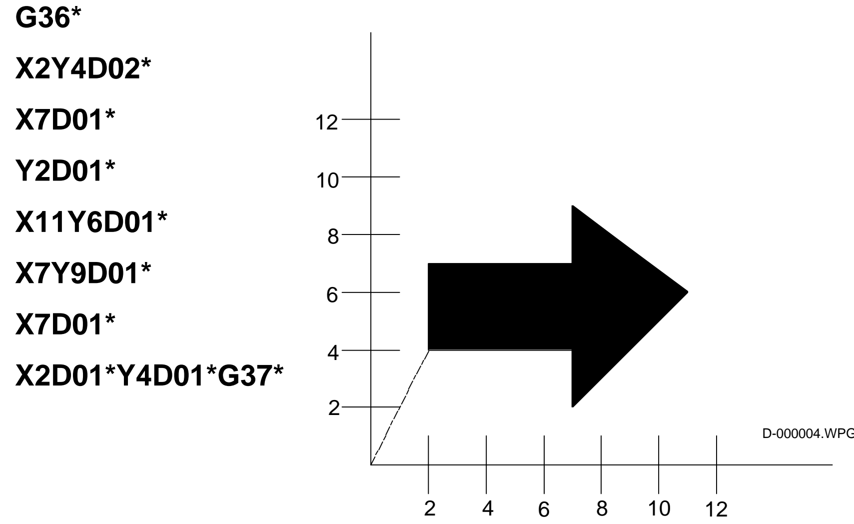


Polygon Area Fill (G36, G37)

G36 and G37 provide a more efficient means of filling closed polygons than stroke fill. When these codes are used, the filled area is defined simply by its closed outline. Stroke fill is an inefficient method of filling a polygon.

G36 turns on polygon area fill. G37 turn it off. There are no variables or apertures. Following a G36 and before G37, all lines drawn with D01 are considered edges of the polygon. D02 closes and fills the polygon.

```
G36*
X2Y4D02*
X7D01*
Y2D01*
X11Y6D01*
X7Y9D01*
X7D01*
X2D01*Y4D01*G37*
```



M Codes

M-codes identify the end of a file. Three M codes are commonly used:

M00	Program stop
M01	Optional stop
M02.	End of program

Glossary

ABSOLUTE POSITION: Position expressed as a distance from the 0,0 point in the data.

APERTURE: Previously, an opening in a wheel through which light passed to expose film. Currently a D code assignment and description of geometry that determines the shape of a feature.

APERTURE MACRO: A mass parameter that describes the geometry of a special aperture and assigns it to a D code.

APERTURE PARAMETER: A mass parameter (AD or AM) that assigns an aperture description to a D code.

CIRCULAR INTERPOLATION: Specifies that the data should be interpreted as arcs; may be single-quadrant or multiquadrant.

COMPOSITE IMAGE: The entire image, including all information layers.

COORDINATE DATA: X,Y position data that describes placement of features in the image.

D CODES: Draft (tool) RS-274D codes. They specify tool exposure action (line draw or flash).

DELIMITER: A character that indicates the beginning and end of a mass parameter.

DIRECTIVE PARAMETER: A mass parameter that controls overall file processing.

EXTENDED GERBER FORMAT: Gerber data that includes mass parameters.

FUNCTION CODES: G codes, D codes, M codes that are part of RS-274D.

G CODES: General function RS-274D codes. They specify interpolation, polygon area fill, etc.

GERBER DATA: Data expressed in Gerber Format.

GERBER FORMAT: A subset of RS-274D Word Address Format that is the universal plotter language; may also contain mass parameters, whose presence make it Extended Gerber Format, or RS-274X.

IMAGE PARAMETER: A parameter that supplies information about an entire image.

KNOCKOUT: A rectangular region about an information layer whose polarity is the opposite of the layer polarity.

LAYER: A named information component of Gerber data that may be treated as a unit, for example, rotated or repeated; has no relationship to a physical PCB layer.

LAYER-SPECIFIC PARAMETER: A mass parameter that applies to a single information layer (for example KO, LN, LP, and SR).

LINEAR INTERPOLATION: Specifies that the data should be interpreted as straight lines.

MASS PARAMETERS: Commands or codes that may be embedded in Gerber Data that specify how the data should be processed.

MULTI-QUADRANT INTERPOLATION: Specifies that the data should be interpreted as arcs that can extend into more than one quadrant, up to 360°).

NEGATIVE: An artwork in which the intended conductive pattern is transparent to light and the areas to be free from conductive material are opaque.

NUMERICAL PRECISION: The number of integer and decimal places used to express a number.

POLARITY: Describes whether the circuitry will be imaged as dark on a clear background (positive) or clear on a dark background (negative). Polarity may be applied to an entire image or to a single layer.

POLYGON AREA FILL: A feature that provides a more efficient means of creating solid (filled) polygons than stroke fill.

RELATIVE POSITION: Position expressed as a distance from the last position.

RS-274D: Electronics Industries Association (EIA) standard data format; a superset of Gerber Format.

RS-274X: Extended Gerber Format, that is, Gerber Format with mass parameters.

SINGLE QUADRANT INTERPOLATION: Specifies that the data should be interpreted as an arc that must fit entire within a single quadrant (90°).

STEP AND REPEAT: A method by which successive exposures of a single image are made to produce a multiple image production master.

STROKE FILL: An inefficient means of creating solid (filled) polygons by "painting" the area.

X DATA: Gerber data that includes mass parameters.

Index

A

absolute data coordinates
  override by IJ....29
Absolute notation....26
ABSOLUTE POSITION....51
absolute values....7
AD....6, 16
AM....6, 16, 19
APERTURE....51
Aperture Defintion....16
Aperture Description....6
Aperture Editor....16
Aperture Macro....6, 19, 51
APERTURE PARAMETER....51
Aperture parameters....5
apertures....5
  special....16
  standard....16
arc....47
arcs....7
AS....5, 25
Assistance....2
Axis assignment....8
Axis Select....5, 25

B

Bulletin Board Service (BBS)....2

C

center image....29
Circle....6, 17, 47, 48
circular interpolation....4, 47, 51
COMPOSITE IMAGE....51
coordinate data....4, 7, 51
  format....26

D

D codes....7, 16, 26, 45, 51
data block....3, 7, 8
  format....15
  maximum length....8
Data types....4
Decimal point programming....26
DELIMITER....51
DIRECTIVE PARAMETER....51
Directive parameters....5

E

Electronic Industries Association....2
end-of-block character....8
end-of-program code....8
English units
  specifying....39
Extended Gerber Format....1, 4, 51

F

file naming conventions....8
File structure....3
Format Statement....4, 5, 26
FS....5, 26
function codes....4, 51
functions codes....10

G

G codes....7, 26, 46, 51
G01, G10, G11, G12....47
G02, G03, G74, G75....47
G36, G37....49
G74....47
G74, G75....48
General File Preparation Guidelines....8
general function codes....46
general functions....7
Gerber data....1, 51
Gerber Format....1, 51
Gerber GPC Aperture Editor.....16
Glossary....51

I

I,J data....7
IF....6, 28
IJ....5, 29
Image Justify....5, 29
Image Name....5, 30
Image Offset....5, 31
IMAGE PARAMETER....5, 51
image placement....29

Image Polarity 5, 32
Image Rotation 5, 33
IN 5, 30
Inches
  specifying 39
Include File 6
incremental data 7
Incremental notation 26
IO 5, 31
IP 5, 32
IR 5, 33

J

justify image 29

K

Knockout 6, 34, 51
KO 6, 34

L

Layer 52
  generated by mass parameters 4, 6
Layer Name 6, 36
Layer Polarity 6, 37
Layers 3
LAYER-SPECIFIC PARAMETER 6, 52
Leading zero omission 26
Line (center) 6, 21
Line (lower left) 6, 22
Line (vector) 6, 21
Linear Interpolation 47, 52
LN 6, 36
LP 6, 37

M

M codes 7, 26, 50
M00 or M02 8
Mass parameters 4, 52
Metric units
  specifying 39
MI 5, 38
Millimeters
  specifying 39
Mirror Image 5, 38
Miscellaneous parameters 6
MO 5, 39
modal. 7
Mode 39
Mode of units 5
Moiré 6, 23
Multiquadrant (360°) Circular Interpolation ... 48, 52

N

N codes 7
Name
  image 30
  layer 36
NEGATIVE 52
NO ZEROES OMITTED 26
notation 26
Numerical precision 8, 52

O

Obround (oval) 17
OF 5, 40
Offset 5, 40
offset image 31
Order of entry
  RS-274D 10
  RS-274X 9
Ordering information 2
Outline 6, 22

P

Parameter delimiter 4, 8
Parameters
  placement of 5
PF 5, 41
Plot Film 5, 41
polarity 34, 52
  image 32
  layer 37
Polygon 6, 22
polygon area fill 1, 49, 52
primitives 19

R

raster device 5
Rectangle or square 17
Regular polygon 18
RELATIVE POSITION 52
Rotate
  image 33
RS-274D 1, 2, 7, 52
  code length 27
  Data Guidelines 10
  order of entry 10
RS-274X 1, 4, 8, 15, 52
  defaults 9
  order of entry 9
  Parameter Guidelines 8
  position in file 9
  required and optional 9

S

Sample Files 10
Scale Factor 5, 42
Sequence number 7, 26
SF 5, 42
Single Quadrant Circular Interpolation 47, 52
Special apertures 16, 19
SR 6, 43
Standard apertures 16, 17
Standard RS-274D Codes 4, 45

Step and Repeat 6, 43, 52
STROKE FILL 52

T

Technical Assistance Center 2
Thermal...6, 23
Trailing zero omission 26

U

Units..39
upper case
    required for entry...8

V

Vector plotters 5

W

Web Page 2
word address format... 4

X

X data... 1, 52
X,Y data.. 7

Z

Zero omission... 26

GERBER SYSTEMS

Part Number 414 100 014 C September 21, 1998

Sun Rasterfile Format
from the man pages :
rasterfile(4) File Formats rasterfile(4)

NAME

rasterfile - Sun's file format for raster images

SYNOPSIS

```
#include <rasterfile.h>
```

DESCRIPTION

A rasterfile is composed of three parts: first, a header containing 8 integers; second, a (possibly empty) set of colormap values; and third, the pixel image, stored a line at a time, in increasing y order. The image is layed out in the file as in a memory pixrect. Each line of the image is rounded up to the nearest 16 bits.

The header is defined by the following structure:

```
struct rasterfile {
     int  ras_magic;
     int  ras_width;
     int  ras_height;
     int  ras_depth;
     int  ras_length;
     int  ras_type;
     int  ras_maptype;
     int  ras_maplength;
};
```

The ras_magic field always contains the following constant:

```
#define   RAS_MAGIC 0x59a66a95
```

The ras_width, ras_height, and ras_depth fields contain the image's width and height in pixels, and its depth in bits per pixel, respectively. The depth is either 1 or 8, corresponding to standard frame buffer depths. The ras_length field contains the length in bytes of the image data. For an unencoded image, this number is computable from the ras_width, ras_height, and ras_depth fields, but for an encoded image it must be explicitly stored in order to be available without decoding the image itself. Note: the length of the header and of the (possibly empty) color-map values are not included in the value of the ras_length field; it is only the image data length. For historical reasons, files of type RT_OLD will usually have a 0 in the ras_length field, and software expecting to encounter such files should be prepared to compute the actual image data length if needed. The ras_maptype and ras_maplength fields contain the type and length in bytes of the colormap values, respectively. If ras_maptype is not RMT_NONE and the ras_maplength is not 0, then the colormap values are the ras_maplength bytes immediately after the header. These values are either uninterpreted bytes (usually with the ras_maptype set to RMT_RAW) or the equal length red, green and blue vectors, in that order (when the ras_maptype is RMT_EQUAL_RGB). In the latter case, the ras_maplength must be three times the size in bytes of any one of the vectors.

SunOS 5.4 Last change: 29 March 1994

Here is an additional note found in the newsgroups :

Inside SUN Rasterfile

Jamie Zawinski
jwz@teak.berkeley.edu

The manpage for rasterfile(5) doesn't say anything about the format of byte-encoded images, or about plane/scanline ordering in multi-plane images.

The first thing in the file is

```
struct rasterfile {
        int ras_magic;
        int ras_width;
        int ras_height;
        int ras_depth;
        int ras_length;
        int ras_type;
        int ras_maptype;
        int ras_maplength;
        };
```

The ras_magic field always contains the following constant:

```
#define RAS_MAGIC 0x59a66a95
```

The ras_length field is the length of the image data (which is the length of the file minus the length of the header and colormap). Catch: this is sometimes zero instead, so you can't really depend on it.

The ras_type field is ras_old=0, ras_standard=1, ras_byte_encoded=2, or ras_experimental=FFFF. There doesn't seem to be any difference between OLD and STANDARD except that the ras_length field is always 0 in OLD.

I didn't deal with cmaps, so from the man page: "The ras_maptype and ras_maplength fields contain the type and length in bytes of the colormap values, respectively. If ras_maptype is not RMT_NONE and the ras_maplength is not 0, then the colormap values are the ras_maplength bytes immediately after the header. These values are either uninterpreted bytes (usually with the ras_maptype set to RMT_RAW) or the equal length red, green and blue vectors, in that order (when the ras_maptype is RMT_EQUAL_RGB). In the latter case, the ras_maplength must be three times the size in bytes of any one of the vectors." Regardless of width, the stored scanlines are rounded up to multiples of 16 bits.

I found the following description of byte-length encoding in Sun-Spots Digest, Volume 6, Issue 84:

The format is composed of many sequences of variable length records. Each record may be 1, 2, or 3 bytes long.

- If the first byte is not 0x80, the record is one byte long, and contains a pixel value. Output 1 pixel of that value.
- If the first byte is 0x80 and the second byte is zero, the record is two bytes long. Output 1 pixel with value 0x80.
- If the first byte is 0x80, and the second byte is not zero, the record is three bytes long. The second byte is a count and the third byte is a value. Output (count+1) pixels of that value.

A run is not terminated at the end of a scan line. So, if there are three lines of red in a picture 100 pixels wide, the first run will be 0x80 0xff 0x<red>, and the second will be 0x80 0x2b 0x<red>.

TAR Format
Intel byte order

Information from File Format List 2.0 by Max Maischein.

--------!-CONTACT_INFO----------------------
If you notice any mistakes or omissions, please let me know! It is only
with YOUR help that the list can continue to grow. Please send
all changes to me rather than distributing a modified version of the list.

This file has been authored in the style of the INTERxxy.* file list
by Ralf Brown, and uses almost the same format.

Please read the file FILEFMTS.1ST before asking me any questions. You may find
that they have already been addressed.

         Max Maischein

Max Maischein, 2:244/1106.17
Max_Maischein@spam.fido.de
corion@informatik.uni-frankfurt.de
Corion on #coders@IRC
--------!-DISCLAIMER------------------------
DISCLAIMER: THIS MATERIAL IS PROVIDED "AS IS". I verify the information
contained in this list to the best of my ability, but I cannot be held
responsible for any problems caused by use or misuse of the information,
especially for those file formats foreign to the PC, like AMIGA or SUN file
formats. If an information it is marked "guesswork" or undocumented, you
should check it carefully to make sure your program will not break with
an unexpected value (and please let me know whether or not it works
the same way).

Information marked with "???" is known to be incomplete or guesswork.

Some file formats were not released by their creators, others are regarded
as proprietary, which means that if your programs deal with them, you might
be looking for trouble. I don't care about this.

The Unix TAR program is an archiver program which stores files in a single
archive without compression.
OFFSET   Count TYPE   Description
@section The Standard Format
A @dfn{tar tape} or file contains a series of records. Each record
contains @code{RECORDSIZE} bytes. Although this format may be
thought of as being on magnetic tape, other media are often used.

Each file archived is represented by a header record which describes
the file, followed by zero or more records which give the contents
of the file. At the end of the archive file there may be a record
filled with binary zeros as an end-of-file marker. A reasonable
system should write a record of zeros at the end, but must not
assume that such a record exists when reading an archive.

The records may be @dfn{blocked} for physical I/O operations. Each
block of @var{N} records (where @var{N} is set by the @samp{-b}
option to @code{tar}) is written with a single @code{write()}
operation. On magnetic tapes, the result of such a write is a
single tape record. When writing an archive, the last block of
records should be written at the full size, with records after the
zero record containing all zeroes. When reading an archive, a
reasonable system should properly handle an archive whose last block
is shorter than the rest, or which contains garbage records after a
zero record.

The header record is defined in C as follows:

```
@example
/*
 * Standard Archive Format - Standard TAR - USTAR
 */
#define  RECORDSIZE  512
#define  NAMSIZ      100
```

```
#define  TUNMLEN      32
#define  TGNMLEN      32

union record @{
    char        charptr[RECORDSIZE];
    struct header @{
        char    name[NAMSIZ];
        char    mode[8];
        char    uid[8];
        char    gid[8];
        char    size[12];
        char    mtime[12];
        char    chksum[8];
        char    linkflag;
        char    linkname[NAMSIZ];
        char    magic[8];
        char    uname[TUNMLEN];
        char    gname[TGNMLEN];
        char    devmajor[8];
        char    devminor[8];
    @} header;
@};

/* The checksum field is filled with this while the checksum is computed. */
#define    CHKBLANKS    "        "        /* 8 blanks, no null */

/* The magic field is filled with this if uname and gname are valid. */
#define    TMAGIC    "ustar  "        /* 7 chars and a null */

/* The magic field is filled with this if this is a GNU format dump entry */
#define    GNUMAGIC  "GNUtar "        /* 7 chars and a null */

/* The linkflag defines the type of file */
#define  LF_OLDNORMAL '\0'       /* Normal disk file, Unix compatible */
#define  LF_NORMAL    '0'        /* Normal disk file */
#define  LF_LINK      '1'        /* Link to previously dumped file */
#define  LF_SYMLINK   '2'        /* Symbolic link */
#define  LF_CHR       '3'        /* Character special file */
#define  LF_BLK       '4'        /* Block special file */
#define  LF_DIR       '5'        /* Directory */
#define  LF_FIFO      '6'        /* FIFO special file */
#define  LF_CONTIG    '7'        /* Contiguous file */

/* Further link types may be defined later. */

/* Bits used in the mode field - values in octal */
#define  TSUID    04000        /* Set UID on execution */
#define  TSGID    02000        /* Set GID on execution */
#define  TSVTX    01000        /* Save text (sticky bit) */

/* File permissions */
#define  TUREAD   00400        /* read by owner */
#define  TUWRITE  00200        /* write by owner */
#define  TUEXEC   00100        /* execute/search by owner */
#define  TGREAD   00040        /* read by group */
#define  TGWRITE  00020        /* write by group */
#define  TGEXEC   00010        /* execute/search by group */
#define  TOREAD   00004        /* read by other */
#define  TOWRITE  00002        /* write by other */
#define  TOEXEC   00001        /* execute/search by other */
@end example
```

All characters in header records are represented by using 8-bit
characters in the local variant of ASCII. Each field within the
structure is contiguous; that is, there is no padding used within
the structure. Each character on the archive medium is stored
contiguously.

Bytes representing the contents of files (after the header record of
each file) are not translated in any way and are not constrained to
represent characters in any character set. The @code{tar} format
does not distinguish text files from binary files, and no
translation of file contents is performed.

The @code{name}, @code{linkname}, @code{magic}, @code{uname}, and
@code{gname} are null-terminated character strings. All other
fileds are zero-filled octal numbers in ASCII. Each numeric field
of width @var{w} contains @var{w}@minus{} 2 digits, a space, and a null,
except @code{size}, and @code{mtime}, which do not contain the
trailing null.

The @code{name} field is the pathname of the file, with directory
names (if any) preceding the file name, separated by slashes.

The @code{mode} field provides nine bits specifying file permissions
and three bits to specify the Set UID, Set GID, and Save Text
(``stick'') modes. Values for these bits are defined above. When
special permissions are required to create a file with a given mode,
and the user restoring files from the archive does not hold such
permissions, the mode bit(s) specifying those special permissions
are ignored. Modes which are not supported by the operating system
restoring files from the archive will be ignored. Unsupported modes
should be faked up when creating or updating an archive; e.g. the
group permission could be copied from the @code{other} permission.

The @code{uid} and @code{gid} fields are the numeric user and group
ID of the file owners, respectively. If the operating system does
not support numeric user or group IDs, these fields should be
ignored.

The @code{size} field is the size of the file in bytes; linked files
are archived with this field specified as zero.
@xref{Extraction Options}; in particular the @samp{-G} option.@refill

The @code{mtime} field is the modification time of the file at the
time it was archived. It is the ASCII representation of the octal
value of the last time the file was modified, represented as an
integer number of seconds since January 1, 1970, 00:00 Coordinated
Universal Time.

The @code{chksum} field is the ASCII representation of the octal
value of the simple sum of all bytes in the header record. Each
8-bit byte in the header is added to an unsigned integer,
initialized to zero, the precision of which shall be no less than
seventeen bits. When calculating the checksum, the @code{chksum}
field is treated as if it were all blanks.

The @code{typeflag} field specifies the type of file archived. If a
particular implementation does not recognize or permit the specified
type, the file will be extracted as if it were a regular file. As
this action occurs, @code{tar} issues a warning to the standard
error.

@table @code
@item LF_NORMAL
@itemx LF_OLDNORMAL
These represent a regular file. In order to be compatible with
older versions of @code{tar}, a @code{typeflag} value of
@code{LF_OLDNORMAL} should be silently recognized as a regular
file. New archives should be created using @code{LF_NORMAL}. Also,
for backward compatibility, @code{tar} treats a regular file whose
name ends with a slash as a directory.

@item LF_LINK
This represents a file linked to another file, of any type,
previously archived. Such files are identified in Unix by each file
having the same device and inode number. The linked-to
name is specified in the @code{linkname} field with a trailing null.

@item LF_SYMLINK
This represents a symbolic link to another file. The linked-to
name is specified in the @code{linkname} field with a trailing null.

@item LF_CHR
@itemx LF_BLK
These represent character special files and block special files

```
respectively.  In this case the @code{devmajor} and @code{devminor}
fields will contain the major and minor device numbers
respectively.  Operating systems may map the device specifications
to their own local specification, or may ignore the entry.

@item LF_DIR
This specifies a directory or sub-directory.  The directory name in
the @code{name} field should end with a slash.  On systems where
disk allocation is performed on a directory basis the @code{size}
field will contain the maximum number of bytes (which may be rounded
to the nearest disk block allocation unit) which the directory may
hold.  A @code{size} field of zero indicates no such limiting.
Systems which do not support limiting in this manner should ignore
the @code{size} field.

@item LF_FIFO
This specifies a FIFO special file.  Note that the archiving of a
FIFO file archives the existence of this file and not its contents.

@item LF_CONTIG
This specifies a contiguous file, which is the same as a normal
file except that, in operating systems which support it,
all its space is allocated contiguously on the disk.  Operating
systems which do not allow contiguous allocation should silently treat
this type as a normal file.

@item 'A' @dots{}
@itemx 'Z'
These are reserved for custom implementations.  Some of these are
used in the GNU modified format, as described below.
@end table

Other values are reserved for specification in future revisions of
the P1003 standard, and should not be used by any @code{tar} program.

The @code{magic} field indicates that this archive was output in the
P1003 archive format.  If this field contains @code{TMAGIC}, the
@code{uname} and @code{gname} fields will contain the ASCII
representation of the owner and group of the file respectively.  If
found, the user and group ID represented by these names will be used
rather than the values within the @code{uid} and @code{gid} fields.

@section GNU Extensions to the Archive Format
The GNU format uses additional file types to describe new types of
files in an archive.  These are listed below.

@table @code
@item LF_DUMPDIR
@itemx 'D'
This represents a directory and a list of files created by the
@samp{-G} option.  The @code{size} field gives the total size of the
associated list of files.  Each filename is preceded by either a @code{'Y'}
(the file should be in this archive) or an @code{'N'} (The file is a
directory, or is not stored in the archive).  Each filename is
terminated by a null.  There is an additional null after the last
filename.

@item LF_MULTIVOL
@itemx 'M'
This represents a file continued from another volume of a
multi-volume archive created with the @samp{-M} option.  The original
type of the file is not given here.  The @code{size} field gives the
maximum size of this piece of the file (assuming the volume does not
end before the file is written out).  The @code{offset} field gives
the offset from the beginning of the file where this part of the
file begins.  Thus @code{size} plus @code{offset} should equal the
original size of the file.

@item LF_VOLHDR
@itemx 'V'
This file type is used to mark the volume header that was given with
the @samp{-V} option when the archive was created.  The @code{name}
field contains the @code{name} given after the @samp{-V} option.
```

```
The @code{size} field is zero.  Only the first file in each volume
of an archive should have this type.

@end table
EXTENSION:
OCCURENCES:
PROGRAMS:
REFERENCE:
SEE ALSO:
VALIDATION:
OFFSET              Count TYPE   Description
0000h                 256 byte   Other header info ?
0100h                   6 char   ID='ustar',0
EXTENSION:TAR
OCCURENCES:PC, Unix
PROGRAMS:TAR
```

An Aldus/Microsoft Technical Memorandum: 8/8/88 Page 1

Preface

This memorandum has been prepared jointly by Aldus and Microsoft in conjunction with leading scanner vendors and other interested parties. This document does not represent a commitment on the part of either Microsoft or Aldus to provide support for this file format in any application. When responding to specific issues raised in this memo, or when requesting additional tag or field assignments, please address your correspondence to either:

Developers_ Desk Windows Marketing Group
Aldus Corporation Microsoft Corporation
411 First Ave. South 16011 NE 36th Way
Suite 200 Box 97017
Seattle, WA 98104 Redmond, WA 98073-9717
(206) 622-5500 (206) 882-8080

Revision Notes

This revision replaces _TIFF Revision 4._ Sections in italics are new or substantially changed in this revision. Also new, but not in italics, are Appendices F, G, and H.

The major enhancements in TIFF 5.0 are:

1. Compression of grayscale and color images, for better disk space utilization. See Appendix F.

2. TIFF Classes_restricted TIFF subsets that can simplify the job of the TIFF implementor. You may wish to scan Appendix G before reading the rest of this document. In fact, you may want to use Appendix G as your main guide, and refer back to the main body of the specification as needed for details concerning TIFF structures and field definitions.

3. Support for _palette color_ images. See the TIFF Class P description in Appendix G, and the new ColorMap field description.

4. Two new tags that can be used to more fully define the characteristics of full color RGB data, and thereby potentially improve the quality of color image reproduction. See Appendix H.

The organization of the document has also changed slightly. In particular, the tags are listed in alphabetical order, within several categories, in the main body of the specification.

TIFF 5.0 page 2

As always, every attempt has been made to add functionality in such a way as to minimize incompatibility problems with older TIFF software. In particular, many TIFF 5.0 files will be readable even by older applications that assume TIFF 4.0 or an earlier version of the specification. One exception is with files that use the new TIFF 5.0 LZW compression scheme. Old applications will have to give up in this case, of course, and will do so _gracefully_ if they have been following the rules.

We are grateful to all of the draft reviewers for their suggestions. Especially helpful were Herb Weiner of Kitchen Wisdom Publishing Company, Brad Pillow of TrueVision, and engineers from Hewlett Packard and Quark. Chris Sears of Magenta Graphics provided information which is included as Appendix H.

Abstract

This document describes TIFF, a tag based file format that is designed to promote the interchange of digital image data.

The fields were defined primarily with desktop publishing and related applications in mind, although it is possible that other sorts of imaging applications may find TIFF to be useful.

The general scenario for which TIFF was invented assumes that applications software for scanning or painting creates a TIFF file, which can then be read and incorporated into a _document_ or _publication_ by an application such as a desktop publishing package.

TIFF is not a printer language or page description language, nor is it intended to be a general document interchange standard. The primary design goal was to provide a rich environment within which the exchange of image data between application programs can be accomplished. This richness is required in order to take advantage of the varying capabilities of scanners and similar devices. TIFF is therefore designed to be a superset of existing image file formats for _desktop_ scanners (and paint programs and anything else that produces images with pixels in them) and will be enhanced on a continuing basis as new capabilities arise. A high priority has been given to structuring the data in such a way as to minimize the pain of future additions. TIFF was designed to be an extensible interchange format.

Although TIFF is claimed to be in some sense a rich format, it can easily be used for simple scanners and applications as well, since the application developer need only be concerned with the capabilities that he requires.

TIFF is intended to be independent of specific operating systems, filing systems, compilers, and processors. The only significant assumption is that the storage medium supports something like a _file,_ defined as a sequence of 8-bit bytes, where the bytes

TIFF 5.0 page 3

are numbered from 0 to N. The largest possible TIFF file is 2**32 bytes in length. Since TIFF uses pointers (byte offsets) quite liberally, a TIFF file is most easily read from a random access device such as a hard disk or flexible diskette, although it should be possible to read and write TIFF files on magnetic

tape.

The recommended MS-DOS, UNIX, and OS/2 file extension for TIFF files is _.TIF._ The recommended Macintosh filetype is _TIFF_. Suggestions for conventions in other computing environments are welcome.

1) Structure

In TIFF, individual fields are identified with a unique tag. This allows particular fields to be present or absent from the file as required by the application. For an explanation of the rationale behind using a tag structure format, see Appendix A.

A TIFF file begins with an 8-byte _image file header_ that points to one or more _image file directories._ The image file directories contain information about the images, as well as pointers to the actual image data.

See Figure 1.

We will now describe these structures in more detail.

Image file header

A TIFF file begins with an 8-byte image file header, containing the following information:

Bytes 0-1: The first word of the file specifies the byte order used within the file. Legal values are:

_II_ (hex 4949)
_MM_ (hex 4D4D)

In the _II_ format, byte order is always from least significant to most significant, for both 16-bit and 32-bit integers. In the _MM_ format, byte order is always from most significant to least significant, for both 16-bit and 32-bit integers. In both formats, character strings are stored into sequential byte locations.

All TIFF readers should support both byte orders_see Appendix G.

Bytes 2-3 The second word of the file is the TIFF _version number._ This number, 42 (2A in hex), is not to be equated with the current Revision of the TIFF specification. In fact, the TIFF version number (42) has never changed, and probably never

TIFF 5.0 page 4

will. If it ever does, it means that TIFF has changed in some way so radical that a TIFF reader should give up immediately. The number 42 was chosen for its deep philosophical significance. It can and should be used as additional verification that this is indeed a TIFF file.

A TIFF file does not have a real version/revision number. This was an explicit, conscious design decision. In many file formats, fields take on different meanings depending on a single version number. The problem is that as the file format _ages,_ it becomes increasingly difficult to document which fields mean

what in a given version, and older software usually has to give up if it encounters a file with a newer version number. We wanted TIFF fields to have a permanent and well-defined meaning, so that _older_ software can usually read _newer_ TIFF files. The bottom line is lower software release costs and more reliable software.

Bytes 4-7 This long word contains the offset (in bytes) of the first Image File Directory. The directory may be at any location in the file after the header but must begin on a word boundary. In particular, an Image File Directory may follow the image data it describes. Readers must simply follow the pointers, wherever they may lead.

(The term _byte offset_ is always used in this document to refer to a location with respect to the beginning of the file. The first byte of the file has an offset of 0.)

Image file directory

An Image File Directory (IFD) consists of a 2-byte count of the number of entries (i.e., the number of fields), followed by a sequence of 12-byte field entries, followed by a 4-byte offset of the next Image File Directory (or 0 if none). Do not forget to write the 4 bytes of 0 after the last IFD.

Each 12-byte IFD entry has the following format:

Bytes 0-1 contain the Tag for the field.
Bytes 2-3 contain the field Type.
Bytes 4-7 contain the Length (_Count_ might have been a better term) of the field.
Bytes 8-11 contain the Value Offset, the file offset (in bytes) of the Value for the field. The Value is expected to begin on a word boundary; the corresponding Value Offset will thus be an even number. This file offset may point to anywhere in the file, including after the image data.

The entries in an IFD must be sorted in ascending order by Tag. Note that this is not the order in which the fields are described in this document. For a numerically ordered list of tags, see

TIFF 5.0 page 5

Appendix E. The Values to which directory entries point need not be in any particular order in the file.

In order to save time and space, the Value Offset is interpreted to contain the Value instead of pointing to the Value if the Value fits into 4 bytes. If the Value is less than 4 bytes, it is left-justified within the 4-byte Value Offset, i.e., stored in the lower-numbered bytes. Whether or not the Value fits within 4 bytes is determined by looking at the Type and Length of the field.

The Length is specified in terms of the data type, not the total number of bytes. A single 16-bit word (SHORT) has a Length of 1, not 2, for example. The data types and their lengths are described below:

1 = BYTE An 8-bit unsigned integer.

2 = ASCII 8-bit bytes that store ASCII codes; the last byte must be null.
3 = SHORT A 16-bit (2-byte) unsigned integer.
4 = LONG A 32-bit (4-byte) unsigned integer.
5 = RATIONAL Two LONG_s: the first represents the numerator of a fraction, the second the denominator.

The value of the Length part of an ASCII field entry includes the null. If padding is necessary, the Length does not include the pad byte. Note that there is no _count byte,_ as there is in Pascal-type strings. The Length part of the field takes care of that. The null is not strictly necessary, but may make things slightly simpler for C programmers.

The reader should check the type to ensure that it is what he expects. TIFF currently allows more than 1 valid type for some fields. For example, ImageWidth and ImageLength were specified as having type SHORT. Very large images with more than 64K rows or columns are possible with some devices even now. Rather than add parallel LONG tags for these fields, it is cleaner to allow both SHORT and LONG for ImageWidth and similar fields. See Appendix G for specific recommendations.

Note that there may be more than one IFD. Each IFD is said to define a _subfile._ One potential use of subsequent subfiles is to describe a _sub-image_ that is somehow related to the main image, such as a reduced resolution version of the image.

If you have not already done so, you may wish to turn to Appendix G to study the sample TIFF images.

2) Definitions

Note that the TIFF structure as described in the previous section is not specific to imaging applications in any way. It is only

TIFF 5.0 page 6

the definitions of the fields themselves that jointly describe an image.

Before we begin defining the fields, we will define some basic concepts. An image is defined to be a rectangular array of _pixels,_ each of which consists of one or more _samples._ With monochromatic data, we have one sample per pixel, and _sample_ and _pixel_ can be used interchangeably. RGB color data contains three samples per pixel.

3) The Fields

This section describes the fields defined in this version of TIFF. More fields may be added in future versions_if possible they will be added in such a way so as not to break old software that encounters a newer TIFF file.

The documentation for each field contains the name of the field (quite arbitrary, but convenient), the Tag value, the field Type, the Number of Values (N) expected, comments describing the field, and the default, if any. Readers must assume the default value if the field does not exist.

No default does not mean that a TIFF writer should not pay attention to the tag. It simply means that there is no default. If the writer has reason to believe that readers will care about the value of this field, the writer should write the field with the appropriate value. TIFF readers can do whatever they want if they encounter a missing _no default_ field that they care about, short of refusing to import the file. For example, if a writer does not write out a PhotometricInterpretation field, some applications will interpret the image _correctly,_ and others will display the image inverted. This is not a good situation, and writers should be careful not to let it happen.

The fields are grouped into several categories: basic, informational, facsimile, document storage and retrieval, and no longer recommended. A future version of the specification may pull some of these categories into separate companion documents.

Many fields are described in this document, but most are not _required._ See Appendix G for a list of required fields, as well as examples of how to combine fields into valid and useful TIFF files.

Basic Fields

Basic fields are fields that are fundamental to the pixel architecture or visual characteristics of an image.

BitsPerSample
Tag = 258 (102)
Type = SHORT

TIFF 5.0 page 7

N = SamplesPerPixel

Number of bits per sample. Note that this tag allows a different number of bits per sample for each sample corresponding to a pixel. For example, RGB color data could use a different number of bits per sample for each of the three color planes. Most RGB files will have the same number of BitsPerSample for each sample. Even in this case, be sure to include all three entries. Writing _8_ when you mean _8,8,8_ sets a bad precedent for other fields.

Default = 1. See also SamplesPerPixel.

ColorMap
Tag = 320 (140)
Type = SHORT
N = 3 * (2**BitsPerSample)

This tag defines a Red-Green-Blue color map for palette color images. The palette color pixel value is used to index into all 3 subcurves. For example, a Palette color pixel having a value of 0 would be displayed according to the 0th entry of the Red, Green, and Blue subcurves.

The subcurves are stored sequentially. The Red entries come first, followed by the Green entries, followed by the Blue entries. The length of each subcurve is 2**BitsPerSample. A ColorMap entry for an 8-bit Palette color image would therefore have 3 * 256 entries. The width of each entry is 16 bits, as

implied by the type of SHORT. 0 represents the minimum intensity, and 65535 represents the maximum intensity. Black is represented by 0,0,0, and white by 65535, 65535, 65535. The purpose of the color map is to act as a _lookup_ table mapping pixel values from 0 to 2**BitsPerSample-1 into RGB triplets.

The ColorResponseCurves field may be used in conjunction with ColorMap to further refine the meaning of the RGB triplets in the ColorMap. However, the ColorResponseCurves default should be sufficient in most cases.

See also PhotometricInterpretation_palette color.

No default. ColorMap must be included in all palette color images.

ColorResponseCurves
Tag = 301 (12D)
Type = SHORT
N = 3 * (2**BitsPerSample)

This tag defines three color response curves, one each for Red, Green and Blue color information. The Red entries come first, followed by the Green entries, followed by the Blue entries. The

length of each subcurve is 2**BitsPerSample, using the BitsPerSample value corresponding to the respective primary. The width of each entry is 16 bits, as implied by the type of SHORT. 0 represents the minimum intensity, and 65535 represents the maximum intensity. Black is represented by 0,0,0, and white by 65535, 65535, 65535. Therefore, a ColorResponseCurve entry for RGB data where each of the samples is 8 bits deep would have 3 * 256 entries, each consisting of a SHORT.

The purpose of the color response curves is to refine the content of RGB color images.

See Appendix H, section VII, for further information.

Default: curves based on the NTSC recommended gamma of 2.2.

Compression
Tag = 259 (103)
Type = SHORT
N = 1

1 = No compression, but pack data into bytes as tightly as possible, with no unused bits except at the end of a row. The bytes are stored as an array of type BYTE, for BitsPerSample <= 8, SHORT if BitsPerSample > 8 and <= 16, and LONG if BitsPerSample > 16 and <= 32. The byte ordering of data >8 bits must be consistent with that specified in the TIFF file header (bytes 0 and 1). _II_ format files will have the least significant bytes preceeding the most significant bytes while _MM_ format files will have the opposite.

If the number of bits per sample is not a power of 2, and you are willing to give up some space for better performance, you may wish to use the next higher power of 2. For example, if your

data can be represented in 6 bits, you may wish to specify that it is 8 bits deep.

Rows are required to begin on byte boundaries. The number of bytes per row is therefore (ImageWidth * SamplesPerPixel * BitsPerSample + 7) / 8, assuming integer arithmetic, for PlanarConfiguration=1. Bytes per row is (ImageWidth * BitsPerSample + 7) / 8 for PlanarConfiguration=2.

Some graphics systems want rows to be word- or double-word-aligned. Uncompressed TIFF rows will need to be copied into word- or double-word-padded row buffers before being passed to the graphics routines in these environments.

2 = CCITT Group 3 1-Dimensional Modified Huffman run length encoding. See Appendix B: _Data Compression -- Scheme 2._ BitsPerSample must be 1, since this type of compression is defined only for bilevel images.

When you decompress data that has been compressed by Compression=2, you must translate white runs into 0_s and black runs into 1_s. Therefore, the normal PhotometricInterpretation for those compression types is 0 (WhiteIsZero). If a reader encounters a PhotometricInterpretation of 1 (BlackIsZero) for such an image, the image should be displayed and printed with black and white reversed.

5 = LZW Compression, for grayscale, mapped color, and full color images. See Appendix F.

32773 = PackBits compression, a simple byte oriented run length scheme for 1-bit images. See Appendix C.

Data compression only applies to raster image data, as pointed to by StripOffsets. All other TIFF information is unaffected.

Default = 1.

GrayResponseCurve
Tag = 291 (123)
Type = SHORT
N = 2**BitsPerSample

The purpose of the gray response curve and the gray units is to provide more exact photometric interpretation information for gray scale image data, in terms of optical density.

The GrayScaleResponseUnits specifies the accuracy of the information contained in the curve. Since optical density is specified in terms of fractional numbers, this tag is necessary to know how to interpret the stored integer information. For example, if GrayScaleResponseUnits is set to 4 (ten-thousandths of a unit), and a GrayScaleResponseCurve number for gray level 4 is 3455, then the resulting actual value is 0.3455. Optical densitometers typically measure densities within the range of 0.0 to 2.0.

If the gray scale response curve is known for the data in the TIFF file, and if the gray scale response of the output device is

known, then an intelligent conversion can be made between the input data and the output device. For example, the output can be made to look just like the input. In addition, if the input image lacks contrast (as can be seen from the response curve), then appropriate contrast enhancements can be made.

The purpose of the gray scale response curve is to act as a _lookup_ table mapping values from 0 to 2**BitsPerSample-1 into specific density values. The 0th element of the GrayResponseCurve array is used to define the gray value for all pixels having a value of 0, the 1st element of the GrayResponseCurve array is used to define the gray value for all pixels having a value of 1, and so on, up to 2**BitsPerSample-1.

TIFF 5.0 page 10

If your data is _really,_ say, 7-bit data, but you are adding a 1-bit pad to each pixel to turn it into 8-bit data, everything still works: If the data is high-order justified, half of your GrayResponseCurve entries (the odd ones, probably) will never be used, but that doesn_t hurt anything. If the data is low-order justified, your pixel values will be between 0 and 127, so make your GrayResponseCurve accordingly. What your curve does from 128 to 255 doesnít matter. Note that low-order justification is probably not a good idea, however, since not all applications look at GrayResponseCurve. Note also that LZW compression yields the same compression ratio regardless of whether the data is high-order or low-order justified.

It is permissable to have a GrayResponseCurve even for bilevel (1-bit) images. The GrayResponseCurve will have 2 values. It should be noted, however, that TIFF B readers are not required to pay attention to GrayResponseCurves in TIFF B files. See Appendix G.

If both GrayResponseCurve and PhotometricInterpretation fields exist in the IFD, GrayResponseCurve values override the PhotometricInterpretation value. But it is a good idea to write out both, since some applications do not yet pay attention to the GrayResponseCurve.

Writers may wish to purchase a Kodak Reflection Density Guide, catalog number 146 5947, available for $10 or so at prepress supply houses, to help them figure out reasonable density values for their scanner or frame grabber. If that sounds like too much work, we recommend a curve that is linear in intensity/reflectance. To compute reflectance from density: R = 1 / pow(10,D). To compute density from reflectance: D = log10 (1/R). A typical 4-bit GrayResponseCurve may look therefore something like: 2000, 1177, 875, 699, 574, 477, 398, 331, 273, 222, 176, 135, 97, 62, 30, 0, assuming GrayResponseUnit=3. Such a curve would be consistent with PhotometricInterpretation=1.

See also GrayResponseUnit, PhotometricInterpretation, ColorMap.

GrayResponseUnit
Tag = 290 (122)
Type = SHORT
N = 1

1 = Number represents tenths of a unit.
2 = Number represents hundredths of a unit.

3 = Number represents thousandths of a unit.
4 = Number represents ten-thousandths of a unit.
5 = Number represents hundred-thousandths of a unit.

Modifies GrayResponseCurve.

See also GrayResponseCurve.

TIFF 5.0 page 11

For historical reasons, the default is 2. However, for greater accuracy, we recommend using 3.

ImageLength
Tag = 257 (101)
Type = SHORT or LONG
N = 1

The image_s length (height) in pixels (Y: vertical). The number of rows (sometimes described as _scan lines") in the image. See also ImageWidth.

No default.

ImageWidth
Tag = 256 (100)
Type = SHORT or LONG
N = 1

The image_s width, in pixels (X: horizontal). The number of columns in the image. See also ImageLength.

No default.

NewSubfileType
Tag = 254 (FE)
Type = LONG
N = 1

Replaces the old SubfileType field, due to limitations in the definition of that field.

A general indication of the kind of data that is contained in this subfile. This field is made up of a set of 32 flag bits. Unused bits are expected to be 0. Bit 0 is the low-order bit.

Currently defined values are:

Bit 0 is 1 if the image is a reduced resolution version of another image in this TIFF file; else the bit is 0.
Bit 1 is 1 if the image is a single page of a multi-page image (see the PageNumber tag description); else the bit is 0.
Bit 2 is 1 if the image defines a transparency mask for another image in this TIFF file. The PhotometricInterpretation value must be 4, designating a transparency mask.

These values have been defined as bit flags because they are pretty much independent of each other. For example, it be useful to have four images in a single TIFF file: a full resolution

image, a reduced resolution image, a transparency mask for the

TIFF 5.0 page 12

full resolution image, and a transparency mask for the reduced resolution image. Each of the four images would have a different value for the NewSubfileType field.

Default is 0.

PhotometricInterpretation
Tag = 262 (106)
Type = SHORT
N = 1

0 = For bilevel and grayscale images: 0 is imaged as white. 2**BitsPerSample-1 is imaged as black. If GrayResponseCurve exists, it overrides the PhotometricInterpretation value, although it is safer to make them match, since some old applications may still be ignoring GrayResponseCurve. This is the normal value for Compression=2.

1 = For bilevel and grayscale images: 0 is imaged as black. 2**BitsPerSample-1 is imaged as white. If GrayResponseCurve exists, it overrides the PhotometricInterpretation value, although it is safer to make them match, since some old applications may still be ignoring GrayResponseCurve. If this value is specified for Compression=2, the image should display and print reversed.

2 = RGB. In the RGB model, a color is described as a combination of the three primary colors of light (red, green, and blue) in particular concentrations. For each of the three samples, 0 represents minimum intensity, and 2**BitsPerSample - 1 represents maximum intensity. Thus an RGB value of (0,0,0) represents black, and (255,255,255) represents white, assuming 8-bit samples. For PlanarConfiguration = 1, the samples are stored in the indicated order: first Red, then Green, then Blue. For PlanarConfiguration = 2, the StripOffsets for the sample planes are stored in the indicated order: first the Red sample plane StripOffsets, then the Green plane StripOffsets, then the Blue plane StripOffsets.

The ColorResponseCurves field may be used to globally refine or alter the color balance of an RGB image without having to change the values of the pixels themselves.

3="Palette color._ In this mode, a color is described with a single sample. The sample is used as an index into ColorMap. The sample is used to index into each of the red, green and blue curve tables to retrieve an RGB triplet defining an actual color. When this PhotometricInterpretation value is used, the color response curves must also be supplied. SamplesPerPixel must be 1.

4 = Transparency Mask. This means that the image is used to define an irregularly shaped region of another image in the same

TIFF 5.0 page 13

TIFF file. SamplesPerPixel and BitsPerSample must be 1. PackBits compression is recommended. The 1-bits define the interior of the region; the 0-bits define the exterior of the region. The Transparency Mask must have the same ImageLength and ImageWidth as the main image.

A reader application can use the mask to determine which parts of the image to display. Main image pixels that correspond to 1-bits in the transparency mask are imaged to the screen or printer, but main image pixels that correspond to 0-bits in the mask are not displayed or printed.

It is possible to generalize the notion of a transparency mask to include partial transparency, but it is not clear that such information would be useful to a desktop publishing program.

No default. That means that if you care if your image is displayed and printed as _normal_ vs _inverted,_ you must write out this field. Do not rely on applications defaulting to what you want! PhotometricInterpretation = 1 is recommended for bilevel (except for Compression=2) and grayscale images, due to popular user interfaces for changing the brightness and contrast of images.

PlanarConfiguration
Tag = 284 (11C)
Type = SHORT
N = 1

1 = The sample values for each pixel are stored contiguously, so that there is a single image plane. See PhotometricInterpretation to determine the order of the samples within the pixel data. So, for RGB data, the data is stored RGBRGBRGB...and so on.

2 = The samples are stored in separate _sample planes._ The values in StripOffsets and StripByteCounts are then arranged as a 2-dimensional array, with SamplesPerPixel rows and StripsPerImage columns. (All of the columns for row 0 are stored first, followed by the columns of row 1, and so on.) PhotometricInterpretation describes the type of data that is stored in each sample plane. For example, RGB data is stored with the Red samples in one sample plane, the Green in another, and the Blue in another.

If SamplesPerPixel is 1, PlanarConfiguration is irrelevant, and should not be included.
Default is 1. See also BitsPerSample, SamplesPerPixel.

Predictor
Tag = 317 (13D)
Type = SHORT

TIFF 5.0 page 14

N   = 1

To be used when Compression=5 (LZW). See Appendix F.

1 = No prediction scheme used before coding.

Default is 1.

ResolutionUnit
Tag = 296 (128)
Type = SHORT
N   = 1

To be used with XResolution and YResolution.

1 = No absolute unit of measurement. Used for images that may have a non-square aspect ratio, but no meaningful absolute dimensions. The drawback of ResolutionUnit=1 is that different applications will import the image at different sizes. Even if the decision is quite arbitrary, it might be better to use dots per inch or dots per centimeter, and pick XResolution and YResolution such that the aspect ratio is correct and the maximum dimension of the image is about four inches (the _four_ is quite arbitrary.)
2 = Inch.
3 = Centimeter.

Default is 2. See also XResolution, YResolution.

RowsPerStrip
Tag = 278 (116)
Type = SHORT or LONG
N   = 1

The number of rows per strip. The image data is organized into strips for fast access to individual rows when the data is compressed_though this field is valid even if the data is not compressed.

RowsPerStrip and ImageLength together tell us the number of strips in the entire image. The equation is StripsPerImage = (ImageLength + RowsPerStrip - 1) / RowsPerStrip, assuming integer arithmetic.

Note that either SHORT or LONG values can be used to specify RowsPerStrip. SHORT values may be used for small TIFF files. It should be noted, however, that earlier TIFF specification revisions required LONG values and that some software may not expect SHORT values. See Appendix G for further recommendations.

Default is 2**32 - 1, which is effectively infinity. That is, the entire image is one strip. We do not recommend a single

TIFF 5.0   page 15

strip, however. Choose RowsPerStrip such that each strip is about 8K bytes, even if the data is not compressed, since it makes buffering simpler for readers. The _8K_ part is pretty arbitrary, but seems to work well.

See also ImageLength, StripOffsets, StripByteCounts.

SamplesPerPixel
Tag = 277 (115)
Type = SHORT
N   = 1

The number of samples per pixel. SamplesPerPixel is 1 for bilevel, grayscale, and palette color images. SamplesPerPixel is 3 for RGB images.

Default = 1. See also BitsPerSample, PhotometricInterpretation.

StripByteCounts
Tag = 279 (117)
Type = SHORT or LONG
N   = StripsPerImage for PlanarConfiguration equal to 1.
     = SamplesPerPixel * StripsPerImage for PlanarConfiguration equal to 2

For each strip, the number of bytes in that strip. The existence of this field greatly simplifies the chore of buffering compressed data, if the strip size is reasonable.

No default. See also StripOffsets, RowsPerStrip.

StripOffsets
Tag = 273 (111)
Type = SHORT or LONG
N   = StripsPerImage for PlanarConfiguration equal to 1.
     = SamplesPerPixel * StripsPerImage for PlanarConfiguration equal to 2

For each strip, the byte offset of that strip. The offset is specified with respect to the beginning of the TIFF file. Note that this implies that each strip has a location independent of the locations of other strips. This feature may be useful for editing applications. This field is the only way for a reader to find the image data, and hence must exist.

Note that either SHORT or LONG values can be used to specify the strip offsets. SHORT values may be used for small TIFF files. It should be noted, however, that earlier TIFF specifications required LONG strip offsets and that some software may not expect SHORT values. See Appendix G for further recommendations.

TIFF 5.0   page 16

No default. See also StripByteCounts, RowsPerStrip.

XResolution
Tag = 282 (11A)
Type = RATIONAL
N   = 1

The number of pixels per ResolutionUnit in the X direction, i.e., in the ImageWidth direction. It is, of course, not mandatory

that the image be actually printed at the size implied by this parameter. It is up to the application to use this information as it wishes.

No default. See also YResolution, ResolutionUnit.

YResolution
Tag = 283 (11B)
Type = RATIONAL
N = 1

The number of pixels per ResolutionUnit in the Y direction, i.e., in the ImageLength direction.

No default. See also XResolution, ResolutionUnit.

Informational Fields

Informational fields are fields that can provide useful information to a user, such as where the image came from. Most are ASCII fields. An application could have some sort of _More Info..._ dialog box to display such information.

Artist
Tag = 315 (13B)
Type = ASCII

Person who created the image.

If you need to attach a Copyright notice to an image, this is the place to do it. In fact, you may wish to write out the contents of the field immediately after the 8-byte TIFF header. Just make sure your IFD and field pointers are set accordingly, and you_re all set.

DateTime
Tag = 306 (132)
Type = ASCII
N = 20

TIFF 5.0 page 17

Date and time of image creation. Use the format _YYYY:MM:DD HH:MM:SS_, with hours on a 24-hour clock, and one space character between the date and the time. The length of the string, including the null, is 20 bytes.

HostComputer
Tag = 316 (13C)
Type = ASCII

ENIAC, or whatever.

See also Make, Model, Software.

ImageDescription

Tag = 270 (10E)
Type = ASCII

For example, a user may wish to attach a comment such as _1988 company picnic_ to an image.

It has been suggested that this is what the newspaper and magazine industry calls a _slug._

Make
Tag = 271 (10F)
Type = ASCII

Manufacturer of the scanner, video digitizer, or whatever.

See also Model, Software.

Model
Tag = 272 (110)
Type = ASCII

The model name/number of the scanner, video digitizer, or whatever.

This tag is intended for user information only.

See also Make, Software.

Software
Tag = 305 (131)
Type = ASCII

Name and release number of the software package that created the image.

TIFF 5.0   page 18

This tag is intended for user information only.

See also Make, Model.

Facsimile Fields

Facsimile fields may be useful if you are using TIFF to store facsimile messages in _raw_ form. They are not recommended for use in interchange with desktop publishing applications.

Compression (a basic tag)
Tag = 259 (103)
Type = SHORT
N   = 1

3 = Facsimile-compatible CCITT Group 3, exactly as specified in _Standardization of Group 3 facsimile apparatus for document transmission,_ Recommendation T.4, Volume VII, Fascicle VII.3, Terminal Equipment and Protocols for Telematic Services, The International Telegraph and Telephone Consultative Committee

(CCITT), Geneva, 1985, pages 16 through 31. Each strip must begin on a byte boundary. (But recall that an image can be a single strip.) Rows that are not the first row of a strip are not required to begin on a byte boundary. The data is stored as bytes, not words_byte-reversal is not allowed. See the Group3Options field for Group 3 options such as 1D vs 2D coding.

4 = Facsimile-compatible CCITT Group 4, exactly as specified in _Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus,_ Recommendation T.6, Volume VII, Fascicle VII.3, Terminal Equipment and Protocols for Telematic Services, The International Telegraph and Telephone Consultative Committee (CCITT), Geneva, 1985, pages 40 through 48. Each strip must begin on a byte boundary. Rows that are not the first row of a strip are not required to begin on a byte boundary. The data is stored as bytes, not words. See the Group4Options field for Group 4 options.

Group3Options
Tag = 292 (124)
Type = LONG
N = 1

See Compression=3. This field is made up of a set of 32 flag bits. Unused bits are expected to be 0. Bit 0 is the low-order bit. It is probably not safe to try to read the file if any bit of this field is set that you don_t know the meaning of.

Bit 0 is 1 for 2-dimensional coding (else 1-dimensional is assumed). For 2-D coding, if more than one strip is specified, each strip must begin with a 1-dimensionally coded line. That

TIFF 5.0 page 19

is, RowsPerStrip should be a multiple of _Parameter K_ as documented in the CCITT specification.

Bit 1 is 1 if uncompressed mode is used.

Bit 2 is 1 if fill bits have been added as necessary before EOL codes such that EOL always ends on a byte boundary, thus ensuring an eol-sequence of a 1 byte preceded by a zero nibble: xxxx-0000 0000-0001.

Default is 0, for basic 1-dimensional coding. See also Compression.

Group4Options
Tag = 293 (125)
Type = LONG
N = 1

See Compression=4. This field is made up of a set of 32 flag bits. Unused bits are expected to be 0. Bit 0 is the low-order bit. It is probably not safe to try to read the file if any bit of this field is set that you don_t know the meaning of. Gray scale and color coding schemes are under study, and will be added when finalized.

For 2-D coding, each strip is encoded as if it were a separate image. In particular, each strip begins on a byte boundary; and

the coding for the first row of a strip is encoded independently of the previous row, using horizontal codes, as if the previous row is entirely white. Each strip ends with the 24-bit end-of-facsimile block (EOFB).

Bit 0 is unused.
Bit 1 is 1 if uncompressed mode is used.

Default is 0, for basic 2-dimensional binary compression. See also Compression.

Document Storage and Retrieval Fields

These fields may be useful for document storage and retrieval applications. They are not recommended for use in interchange with desktop publishing applications.

DocumentName
Tag = 269 (10D)
Type = ASCII

The name of the document from which this image was scanned.

See also PageName.

TIFF 5.0 page 20

PageName
Tag = 285 (11D)
Type = ASCII

The name of the page from which this image was scanned.

See also DocumentName.

No default.

PageNumber
Tag = 297 (129)
Type = SHORT
N = 2

This tag is used to specify page numbers of a multiple page (e.g. facsimile) document. Two SHORT values are specified. The first value is the page number; the second value is the total number of pages in the document.

Note that pages need not appear in numerical order. The first page is 0 (zero).

No default.

XPosition
Tag = 286 (11E)
Type = RATIONAL

The X offset of the left side of the image, with respect to the left side of the page, in ResolutionUnits.

No default. See also YPosition.

YPosition
Tag = 287 (11F)
Type = RATIONAL

The Y offset of the top of the image, with respect to the top of the page, in ResolutionUnits. In the TIFF coordinate scheme, the positive Y direction is down, so that YPosition is always positive.

No default. See also XPosition.

No Longer Recommended

TIFF 5.0 page 21

These fields are not recommended except perhaps for local use. They should not be used for image interchange. They have either been superseded by other fields, have been found to have serious drawbacks, or are simply not as useful as once thought. They may be dropped entirely from a future revision of the specification.

CellLength
Tag = 265 (109)
Type = SHORT
N = 1

The length, in 1-bit samples, of the dithering/halftoning matrix. Assumes that Threshholding = 2.

This field, plus CellWidth and Threshholding, are problematic because they cannot safely be used to reverse-engineer grayscale image data out of dithered/halftoned black-and-white data, which is their only plausible purpose. The only _right_ way to do it is to not bother with anything like these fields, and instead write some sophisticated pattern-matching software that can handle screen angles that are not multiples of 45 degrees, and other such challenging dithered/halftoned data.

So we do not recommend trying to convert dithered or halftoned data into grayscale data. Dithered and halftoned data require careful treatment to avoid _stretch marks,_ but it can be done. If you want grayscale images, get them directly from the scanner or frame grabber or whatever.

No default. See also Threshholding.

CellWidth
Tag = 264 (108)
Type = SHORT
N = 1

The width, in 1-bit samples, of the dithering/halftoning matrix.

No default. See also Threshholding. See the comments for

CellLength.

FillOrder
Tag = 266 (10A)
Type = SHORT
N = 1

The order of data values within a byte.
1 = most significant bits of the byte are filled first. That is, data values (or code words) are ordered from high order bit to low order bit within a byte.
2 = least significant bits are filled first. Since little interest has been expressed in least-significant fill order to

date, and since it is easy and inexpensive for writers to reverse bit order (use a 256-byte lookup table), we recommend FillOrder=2 for private (non-interchange) use only.

Default is FillOrder = 1.

FreeByteCounts
Tag = 289 (121)
Type = LONG

For each _free block_ in the file, the number of bytes in the block.

TIFF readers can ignore FreeOffsets and FreeByteCounts if present.

FreeOffsets and FreeByteCounts do not constitute a remapping of the logical address space of the file.

Since this information can be generated by scanning the IFDs, StripOffsets, and StripByteCounts, FreeByteCounts and FreeOffsets are not needed.

In addition, it is not clear what should happen if FreeByteCounts and FreeOffsets exist in more than one IFD.

See also FreeOffsets.

FreeOffsets
Tag = 288 (120)
Type = LONG

For each _free block_ in the file, its byte offset.

See also FreeByteCounts.

MaxSampleValue
Tag = 281 (119)
Type = SHORT
N = SamplesPerPixel

The maximum used sample value. For example, if the image consists of 6-bit data low-order-justified into 8-bit bytes,

MaxSampleValue will be no greater than 63. This is field is not to be used to affect the visual appearance of the image when displayed. Nor should the values of this field affect the interpretation of any other field. Use it for statistical purposes only.

Default is 2**(BitsPerSample) - 1.

TIFF 5.0 page 23

MinSampleValue
Tag = 280 (118)
Type = SHORT
N = SamplesPerPixel

The minimum used sample value. This field is not to be used to affect the visual appearance of the image when displayed. See the comments for MaxSampleValue.

Default is 0.

SubfileType
Tag = 255 (FF)
Type = SHORT
N = 1

A general indication of the kind of data that is contained in this subfile. Currently defined values are:

1 = full resolution image data_ImageWidth, ImageLength, and StripOffsets are required fields; and
2 = reduced resolution image data_ImageWidth, ImageLength, and StripOffsets are required fields. It is further assumed that a reduced resolution image is a reduced version of the entire extent of the corresponding full resolution data.
3 = single page of a multi-page image (see the PageNumber tag description).

Note that several image types can be found in a single TIFF file, with each subfile described by its own IFD.

No default.

Continued use of this field is not recommended. Writers should instead use the new and more general NewSubfileType field.

Orientation
Tag = 274 (112)
Type = SHORT
N = 1

1 = The 0th row represents the visual top of the image, and the 0th column represents the visual left hand side.
2 = The 0th row represents the visual top of the image, and the 0th column represents the visual right hand side.
3 = The 0th row represents the visual bottom of the image, and the 0th column represents the visual right hand side.
4 = The 0th row represents the visual bottom of the image, and the 0th column represents the visual left hand side.

5 = The 0th row represents the visual left hand side of the image, and the 0th column represents the visual top.

TIFF 5.0 page 24

6 = The 0th row represents the visual right hand side of the image, and the 0th column represents the visual top.
7 = The 0th row represents the visual right hand side of the image, and the 0th column represents the visual bottom.
8 = The 0th row represents the visual left hand side of the image, and the 0th column represents the visual bottom.

Default is 1.

This field is recommended for private (non-interchange) use only. It is extremely costly for most readers to perform image rotation _on the fly,_ i.e., when importing and printing; and users of most desktop publishing applications do not expect a file imported by the application to be altered permanently in any way.

Threshholding
Tag = 263 (107)
Type = SHORT
N = 1

1 = a bilevel _line art_ scan. BitsPerSample must be 1.
2 = a _dithered_ scan, usually of continuous tone data such as photographs. BitsPerSample must be 1.
3 = Error Diffused.

Default is Threshholding = 1. See also CellWidth, CellLength.
4) Private Fields

An organization may wish to store information that is meaningful to only that organization in a TIFF file. Tags numbered 32768 or higher are reserved for that purpose. Upon request, the administrator will allocate and register a block of private tags for an organization, to avoid possible conflicts with other organizations. Tags are normally allocated in blocks of five. If that is not enough, feel free to ask for more. You do not need to tell the TIFF administrator or anyone else what you are going to use them for.

Private enumerated values can be accommodated in a similar fashion. For example, you may wish to experiment with a new compression scheme within TIFF. Enumeration constants numbered 32768 or higher are reserved for private usage. Upon request, the administrator will allocate and register a block of enumerated values for a particular field (Compression, in our example), to avoid possible conflicts.

Tags and values which are allocated in the private number range are not prohibited from being included in a future revision of this specification. Several such instances can be found in the TIFF specification.

Do not choose your own tag numbers. If you do, it could cause serious problems some day.

TIFF 5.0 page 25

5) Image File Format Issues

In the quest to give users no reason NOT to buy a product, some scanning and image editing applications overwhelm users with an incredible number of _Save As..._ options. Let_s get rid of as many of these as we possibly can. For example, a single TIFF choice should suffice once most major readers are supporting the three TIFF compression schemes; then writers can always compress. And given TIFF_s flexibility, including private tag and image editing support features, there does not seem to be any legitimate reason for continuing to write image files using proprietary formats.

Along the same lines, there is no excuse for making a user have to know the file format of a file that is to be read by an application program. TIFF files, as well as most other file formats, contain sufficient information to enable software to automatically and reliably distinguish one type of file from another.

6) For Further Information

Contact the Aldus Developers_ Desk for sample TIFF files, source code fragments, and a list of features that are currently supported in Aldus products. The Aldus Developers_ Desk is the current _TIFF administrator._

Various TIFF related aids are found in Microsoft_s Windows Developers Tookit for developers writing Windows applications.

Finally, a number of scanner vendors are providing various TIFF services, such as helping to distribute the TIFF specification and answering TIFF questions. Contact the appropriate product manager or developer support service group.

Appendix A: Tag Structure Rationale

A file format is defined by both form (structure) and content. The content of TIFF consists of definitions of individual fields. It is therefore the content that we are ultimately interested in. The structure merely tells us how to find the fields. Yet the structure deserves serious consideration for a number of reasons that are not at all obvious at first glance. Since the structure described herein departs significantly from several other approaches, it may be useful to discuss the rationale behind it.

The simplest, most straightforward structure for something like an image file is a positional format. In a positional scheme, the location of the data defines what the data means. For example, the field for _number of rows_ might begin at byte offset 30 in the image file.

This approach is simple and easy to implement and is perfect for static environments. But if a significant amount of ongoing change must be accommodated, subtle problems begin to appear. For example, suppose that a field must be superseded by a new, more general field. You could bump a version number to flag the change. Then new software has no problem doing something sensible with old data, and all old software will reject the new data, even software that didn_t care about the old field. This may seem like no more than a minor annoyance at first glance, but causing old software to break more often than it would really need to can be very costly and, inevitably, causes much gnashing of teeth among customers.

Furthermore, it can be avoided. One approach is to store a _valid_ flag bit for each field. Now you don_t have to bump the version number, as long as you can put the new field somewhere that doesn_t disturb any of the old fields. Old software that didn_t care about that old field anyway can continue to function. (Old software that did care will of course have to give up, but this is an unavoidable price to be paid for the sake of progress, barring total omniscience.)

Another problem that crops up frequently is that certain fields are likely to make sense only if other fields have certain values. This is not such a serious problem in practice; it just makes things more confusing. Nevertheless, we note that the _valid_ flag bits described in the previous paragraph can help to clarify the situation.

Field-dumping programs can be very helpful for diagnostic purposes. A desirable characteristic of such a program is that it doesn_t have to know much about what it is dumping. In particular, it would be nice if the program could dump ASCII data in ASCII format, integer data in integer format, and so on, without having to teach the program about new fields all the time. So maybe we should add a _data type_ component to our

TIFF 5.0 page 27

fields, plus information on how long the field is, so that our dump program can walk through the fields without knowing what the fields _mean."

But note that if we add one more component to each field, namely a tag that tells what the field means, we can dispense with the _valid_ flag bits, and we can also avoid wasting space on the non-valid fields in the file. Simple image creation applications can write out several fields and be done.

We have now derived the essentials of a tag-based image file format.

Finally, a caveat. A tag based scheme cannot guarantee painless growth. But is does provide a useful tool to assist in the process.

TIFF 5.0 page 28

Appendix B: Data Compression_Scheme 2

Abstract

This document describes a method for compressing bilevel data that is based on the CCITT Group 3 1D facsimile compression scheme.

References

1. _Standardization of Group 3 facsimile apparatus for document transmission,_ Recommendation T.4, Volume VII, Fascicle VII.3, Terminal Equipment and Protocols for Telematic Services, The International Telegraph and Telephone Consultative Committee (CCITT), Geneva, 1985, pages 16 through 31.
2. _Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus,_ Recommendation T.6, Volume VII, Fascicle VII.3, Terminal Equipment and Protocols for Telematic Services, The International Telegraph and Telephone Consultative Committee (CCITT), Geneva, 1985, pages 40 through 48.

We do not believe that these documents are necessary in order to implement Compression=2. We have included (verbatim in most places) all the pertinent information in this Appendix. However, if you wish to order the documents, you can write to ANSI, Attention: Sales, 1430 Broadway, New York, N.Y., 10018. Ask for the publication listed above_it contains both Recommendation T.4 and T.6.

Relationship to the CCITT Specifications

The CCITT Group 3 and Group 4 specifications describe communications protocols for a particular class of devices. They are not by themselves sufficient to describe a disk data format. Fortunately, however, the CCITT coding schemes can be readily adapted to this different environment. The following is one such adaptation. Most of the language is copied directly from the CCITT specifications.

Coding Scheme

A line (row) of data is composed of a series of variable length code words. Each code word represents a run length of either all white or all black. (Actually, more than one code word may be required to code a given run, in a manner described below.) White runs and black runs alternate.

In order to ensure that the receiver (decompressor) maintains color synchronization, all data lines will begin with a white run length code word set. If the actual scan line begins with a

TIFF 5.0 page 29

black run, a white run length of zero will be sent (written). Black or white run lengths are defined by the code words in Tables 1 and 2. The code words are of two types: Terminating code words and Make-up code words. Each run length is represented by zero or more Make-up code words followed by exactly one Terminating code word.

Run lengths in the range of 0 to 63 pels (pixels) are encoded with their appropriate Terminating code word. Note that there is a different list of code words for black and white run lengths.

Run lengths in the range of 64 to 2623 (2560+63) pels are encoded first by the Make-up code word representing the run length that is nearest to, not longer than, that required. This is then followed by the Terminating code word representing the difference

between the required run length and the run length represented by the Make-up code.

Run lengths in the range of lengths longer than or equal to 2624 pels are coded first by the Make-up code of 2560. If the remaining part of the run (after the first Make-up code of 2560) is 2560 pels or greater, additional Make-up code(s) of 2560 are issued until the remaining part of the run becomes less than 2560 pels. Then the remaining part of the run is encoded by Terminating code or by Make-up code plus Terminating code, according to the range mentioned above.

It is considered an unrecoverable error if the sum of the run lengths for a line does not equal the value of the ImageWidth field.

New rows always begin on the next available byte boundary.

No EOL code words are used. No fill bits are used, except for the ignored bits at the end of the last byte of a row. RTC is not used.

Table 1/T.4 Terminating codes

White run length	Code word	Black run length	Code word
0	00110101	0	0000110111
1	000111	1	010
2	0111	2	11
3	1000	3	10
4	1011	4	011
5	1100	5	0011
6	1110	6	0010
7	1111	7	00011
8	10011	8	000101
9	10100	9	000100
10	00111	10	0000100
11	01000	11	0000101
12	001000	12	0000111
13	000011	13	00000100
14	110100	14	00000111
15	110101	15	000011000
16	101010	16	0000010111
17	101011	17	0000011000
18	0100111	18	0000001000
19	0001100	19	00001100111
20	0001000	20	00001101000
21	0010111	21	00001101100
22	0000011	22	00000110111
23	0000100	23	00000101000
24	0101000	24	00000010111
25	0101011	25	00000011000
26	0010011	26	000011001010
27	0100100	27	000011001011
28	0011000	28	000011001100

TIFF 5.0 page 30

29	00000010	29	000011001101
30	00000011	30	000001101000
31	00011010	31	000001101001
32	00011011	32	000001101010
33	00010010	33	000001101011
34	00010011	34	000011010010
35	00010100	35	000011010011
36	00010101	36	000011010100
37	00010110	37	000011010101
38	00010111	38	000011010110
39	00101000	39	000011010111
40	00101001	40	000001101100
41	00101010	41	000001101101
42	00101011	42	000011011010
43	00101100	43	000011011011
44	00101101	44	000001010100
45	00000100	45	000001010101
46	00000101	46	000001010110
47	00001010	47	000001010111
48	00001011	48	000001100100
49	01010010	49	000001100101
50	01010011	50	000001010010
51	01010100	51	000001010011
52	01010101	52	000000100100
53	00100100	53	000000110111
54	00100101	54	000000111000
55	01011000	55	000000100111
56	01011001	56	000000101000
57	01011010	57	000001011000
58	01011011	58	000001011001
59	01001010	59	000000101011
60	01001011	60	000000101100
61	00110010	61	000001011010

TIFF 5.0 page 31

62	00110011	62	000001100110
63	00110100	63	000001100111

Table 2/T.4 Make-up codes

White run length	Code word	Black run length	Code word
64	11011	64	0000001111
128	10010	128	000011001000
192	010111	192	000011001001
256	0110111	256	000001011011
320	00110110	320	000000110011
384	00110111	384	000000110100
448	01100100	448	000000110101
512	01100101	512	0000001101100
576	01101000	576	0000001101101
640	01100111	640	0000001001010
704	011001100	704	0000001001011
768	011001101	768	0000001001100
832	011010010	832	0000001001101
896	011010011	896	0000001110010

960	011010100	960	0000001110011
1024	011010101	1024	0000001110100
1088	011010110	1088	0000001110101
1152	011010111	1152	0000001110110
1216	011011000	1216	0000001110111
1280	011011001	1280	0000001010010
1344	011011010	1344	0000001010011
1408	011011011	1408	0000001010100
1472	010011000	1472	0000001010101
1536	010011001	1536	0000001011010
1600	010011010	1600	0000001011011
1664	011000	1664	0000001100100
1728	010011011	1728	0000001100101
EOL	000000000001	EOL	000000000001

Additional make-up codes

White
and
Black   Make-up
run code
length   word
------   ----

TIFF 5.0 page 32

1792	00000001000
1856	00000001100
1920	00000001101
1984	000000010010
2048	000000010011
2112	000000010100
2176	000000010101
2240	000000010110
2304	000000010111
2368	000000011100
2432	000000011101
2496	000000011110
2560	000000011111

TIFF 5.0 page 33

Appendix C: Data Compression_Scheme 32773_ _PackBits_

Abstract

This document describes a simple compression scheme for bilevel scanned and paint type files.

Motivation

The TIFF specification defines a number of compression schemes. Compression type 1 is really no compression, other than basic pixel packing. Compression type 2, based on CCITT 1D compression, is powerful, but not trivial to implement. Compression type 5 is typically very effective for most bilevel images, as well as many deeper images such as palette color and grayscale images, but is also not trivial to implement. PackBits is a simple but often effective alternative.

Description

Several good schemes were already in use in various settings. We somewhat arbitrarily picked the Macintosh PackBits scheme. It is byte oriented, so there is no problem with word alignment. And it has a good worst case behavior (at most 1 extra byte for every 128 input bytes). For Macintosh users, there are toolbox utilities PackBits and UnPackBits that will do the work for you, but it is easy to implement your own routines.

A pseudo code fragment to unpack might look like this:

Loop until you get the number of unpacked bytes you are expecting:
     Read the next source byte into n.
     If n is between 0 and 127 inclusive, copy the next n+1 bytes literally.

Else if n is between -127 and -1 inclusive, copy the next byte -n+1 times.
Else if n is 128, noop.
Endloop

In the inverse routine, it_s best to encode a 2-byte repeat run as a replicate run except when preceded and followed by a literal run, in which case it_s best to merge the three into one literal run. Always encode 3-byte repeats as replicate runs.

So that_s the algorithm. Here are some other rules:

— Each row must be packed separately. Do not compress across row boundaries.

TIFF 5.0 page 34

— The number of uncompressed bytes per row is defined to be (ImageWidth + 7) / 8. If the uncompressed bitmap is required to have an even number of bytes per row, decompress into word-aligned buffers.
— If a run is larger than 128 bytes, simply encode the remainder of the run as one or more additional replicate runs.

When PackBits data is uncompressed, the result should be interpreted as per compression type 1 (no compression).

TIFF 5.0 page 35

Appendix D

Appendix D has been deleted. It formerly contained guidelines for passing TIFF files on the Microsoft Windows Clipboard. This was judged to not be a good idea, in light of the ever-increasing size of scanned images. Applications are instead encouraged to employ file-based mechanisms to exchange TIFF data. Aldus_ PageMaker, for example, implements a _File Place_ command to allow TIFF files to be imported.

TIFF 5.0 page 36

Appendix E: Numerical List of TIFF Tags

NewSubfileType
Tag = 254 (FE)
Type = LONG
N = 1

SubfileType
Tag = 255 (FF)
Type = SHORT
N = 1

ImageWidth
Tag = 256 (100)
Type = SHORT or LONG
N = 1

ImageLength
Tag = 257 (101)
Type = SHORT or LONG
N = 1

BitsPerSample
Tag = 258 (102)
Type = SHORT
N = SamplesPerPixel

Compression
Tag = 259 (103)
Type = SHORT
N = 1

PhotometricInterpretation
Tag = 262 (106)
Type = SHORT
N = 1

Threshholding
Tag = 263 (107)
Type = SHORT
N = 1

CellWidth
Tag = 264 (108)
Type = SHORT
N = 1

CellLength
Tag = 265 (109)
Type = SHORT
N = 1

TIFF 5.0 page 37

FillOrder
Tag = 266 (10A)
Type = SHORT
N = 1

DocumentName
Tag = 269 (10D)
Type = ASCII

ImageDescription
Tag = 270 (10E)
Type = ASCII

Make
Tag = 271 (10F)
Type = ASCII

Model
Tag = 272 (110)
Type = ASCII

StripOffsets
Tag = 273 (111)
Type = SHORT or LONG
N = StripsPerImage for PlanarConfiguration equal to 1.
= SamplesPerPixel * StripsPerImage for PlanarConfiguration equal to 2

Orientation
Tag = 274 (112)
Type = SHORT
N = 1

SamplesPerPixel
Tag = 277 (115)
Type = SHORT
N = 1

RowsPerStrip
Tag = 278 (116)
Type = SHORT or LONG
N = 1

StripByteCounts
Tag = 279 (117)
Type = LONG or SHORT
N = StripsPerImage for PlanarConfiguration equal to 1.
= SamplesPerPixel * StripsPerImage for PlanarConfiguration equal to 2.

MinSampleValue
Tag = 280 (118)
Type = SHORT
N = SamplesPerPixel

TIFF 5.0 page 38

```
MaxSampleValue
Tag  = 281  (119)
Type = SHORT
N    = SamplesPerPixel

XResolution
Tag  = 282  (11A)
Type = RATIONAL
N    = 1

YResolution
Tag  = 283  (11B)
Type = RATIONAL
N    = 1

PlanarConfiguration
Tag  = 284  (11C)
Type = SHORT
N    = 1

PageName
Tag  = 285  (11D)
Type = ASCII


XPosition
Tag  = 286  (11E)
Type = RATIONAL

YPosition
Tag  = 287  (11F)
Type = RATIONAL

FreeOffsets
Tag  = 288  (120)
Type = LONG

FreeByteCounts
Tag  = 289  (121)
Type = LONG

GrayResponseUnit
Tag  = 290 (122)
Type = SHORT
N    = 1

GrayResponseCurve
Tag  = 291 (123)
Type = SHORT
N    = 2**BitsPerSample

Group3Options
Tag  = 292  (124)
```


TIFF 5.0 page 39

```
Type = LONG
N    = 1
```

```
Group4Options
Tag  =  293  (125)
Type = LONG
N    = 1

ResolutionUnit
Tag  = 296 (128)
Type = SHORT
N    = 1

PageNumber
Tag  = 297  (129)
Type = SHORT
N    = 2

ColorResponseCurves
Tag  = 301 (12D)
Type = SHORT
N    = 3 * (2**BitsPerSample)

Software
Tag  = 305  (131)
Type = ASCII

DateTime
Tag  = 306  (132)
Type = ASCII
N    = 20

Artist
Tag  = 315  (13B)
Type = ASCII

HostComputer
Tag  = 316  (13C)
Type = ASCII

Predictor
Tag  = 317 (13D)
Type = SHORT
N    = 1

WhitePoint
Tag  = 318 (13E)
Type = RATIONAL
N    = 2

PrimaryChromaticities
Tag  = 319 (13F)
Type = RATIONAL
N    = 6
```


TIFF 5.0 page 40

```
ColorMap
Tag  = 320 (140)
Type = SHORT
N    = 3 * (2**BitsPerSample)
```

TIFF 5.0 page 41

Appendix F: Data Compression_Scheme 5_LZW
Compression

Abstract

This document describes an adaptive compression scheme for raster images.

Reference

Terry A. Welch, _A Technique for High Performance Data Compression_, IEEE Computer, vol. 17 no. 6 (June 1984).

Describes the basic Lempel-Ziv & Welch (LZW) algorithm. The author_s goal in the article is to describe a hardware-based compressor that could be built into a disk controller or database engine, and used on all types of data. There is no specific discussion of raster images. We intend to give sufficient information in this Appendix so that the article is not required reading.

Requirements

A compression scheme with the following characteristics should work well in a desktop publishing environment:

— Must work well for images of any bit depth, including images deeper than 8 bits per sample.
— Must be effective: an average compression ratio of at least 2:1 or better. And it must have a reasonable worst-case behavior, in case something really strange is thrown at it.
— Should not depend on small variations between pixels. Palette color images tend to contain abrupt changes in index values, due to common patterning and dithering techniques. These abrupt changes do tend to be repetitive, however, and the scheme should make use of this fact.
— For images generated by paint programs, the scheme should not depend on a particular pattern width. 8x8 pixel patterns are common now, but we should not assume that this situation will not change.
— Must be fast. It should not take more than 5 seconds to decompress a 100K byte grayscale image on a 68020- or 386-based computer. Compression can be slower, but probably not by more than a factor of 2 or 3.
— The level of implementation complexity must be reasonable. We would like something that can be implemented in no more than a couple of weeks by a_competent software engineer with some experience in image processing. The compiled code for compression and decompression combined should be no more than about 10K.
— Does not require floating point software or hardware.

TIFF 5.0 page 42

The following sections describe an algorithm based on the _LZW_ (Lempel-Ziv & Welch) technique that meets the above requirements. In addition meeting our requirements, LZW has the following characteristics:

— LZW is fully reversible. All information is preserved. But if noise or information is removed from an image, perhaps by smoothing or zeroing some low-order bitplanes, LZW compresses images to a smaller size. Thus, 5-bit, 6-bit, or 7-bit data masquerading as 8-bit data compresses better than true 8-bit data. Smooth images also compress better than noisy images, and simple images compress better than complex images.
Ì On a 68082- or 386-based computer, LZW software can be written to compress at between 30K and 80K bytes per second, depending on image characteristics. LZW decompression speeds are typically about 50K bytes per second.
Ì LZW works well on bilevel images, too. It always beats PackBits, and generally ties CCITT 1D (Modified Huffman) compression, on our test images. Tying CCITT 1D is impressive in

that LZW seems to be considerably faster than CCITT 1D, at least in our implementation.
Ì Our implementation is written in C, and compiles to about 2K bytes of object code each for the compressor and decompressor.
Ì One of the nice things about LZW is that it is used quite widely in other applications such as archival programs, and is therefore more of a known quantity.

The Algorithm

Each strip is compressed independently. We strongly recommend that RowsPerStrip be chosen such that each strip contains about 8K bytes before compression. We want to keep the strips small enough so that the compressed and uncompressed versions of the strip can be kept entirely in memory even on small machines, but large enough to maintain nearly optimal compression ratios.

The LZW algorithm is based on a translation table, or string table, that maps strings of input characters into codes. The TIFF implementation uses variable-length codes, with a maximum code length of 12 bits. This string table is different for every strip, and, remarkably, does not need to be kept around for the decompressor. The trick is to make the decompressor automatically build the same table as is built when compressing the data. We use a C-like pseudocode to describe the coding scheme:

```
     InitializeStringTable();
     WriteCode(ClearCode);
     _ = the empty string;
     for each character in the strip {
          K = GetNextCharacter();
          if _+K is in the string table {
```


TIFF 5.0 page 43

```
               _ = _+K;  /* string concatenation */
          } else {
               WriteCode (CodeFromString(_));
               AddTableEntry(_+K);
               _ = K;
          }
     } /* end of for loop */
     WriteCode (CodeFromString(_));
     WriteCode (EndOfInformation);
```

That_s it. The scheme is simple, although it is fairly challenging to implement efficiently. But we need a few explanations before we go on to decompression.

The _characters_ that make up the LZW strings are bytes containing TIFF uncompressed (Compression=1) image data, in our implementation. For example, if BitsPerSample is 4, each 8-bit LZW character will contain two 4-bit pixels. If BitsPerSample is 16, each 16-bit pixel will span two 8-bit LZW characters.

(It is also possible to implement a version of LZW where the LZW character depth equals BitsPerSample, as was described by Draft 2 of Revision 5.0. But there is a major problem with this approach. If BitsPerSample is greater than 11, we can not use 12-bit-maximum codes, so that the resulting LZW table is unacceptably large. Fortunately, due to the adaptive nature of

LZW, we do not pay a significant compression ratio penalty for combining several pixels into one byte before compressing. For example, our 4-bit sample images compressed about 3 percent worse, and our 1-bit images compressed about 5 percent better. And it is easier to write an LZW compressor that always uses the same character depth than it is to write one which can handle varying depths.)

We can now describe some of the routine and variable references in our pseudocode:

InitializeStringTable() initializes the string table to contain all possible single-character strings. There are 256 of them, numbered 0 through 255, since our characters are bytes.

WriteCode() writes a code to the output stream. The first code written is a Clear code, which is defined to be code #256.

_ is our _prefix string._

GetNextCharacter() retrieves the next character value from the input stream. This will be number between 0 and 255, since our characters are bytes.

The _+_ signs indicate string concatenation.

AddTableEntry() adds a table entry. (InitializeStringTable() has already put 256 entries in our table. Each entry consists of a

TIFF 5.0 page 44

single-character string, and its associated code value, which is, in our application, identical to the character itself. That is, the 0th entry in our table consists of the string <0>, with corresponding code value of <0>, the 1st entry in the table consists of the string <1>, with corresponding code value of <1>, ..., and the 255th entry in our table consists of the string <255>, with corresponding code value of <255>.) So the first entry that we add to our string table will be at position 256, right? Well, not quite, since we will reserve code #256 for a special _Clear_ code, and code #257 for a special _EndOfInformation_ code that we will write out at the end of the strip. So the first multiple-character entry added to the string table will be at position 258.

Let_s try an example. Suppose we have input data that looks like:

Pixel 0: <7>
Pixel 1: <7>
Pixel 2: <7>
Pixel 3: <8>
Pixel 4: <8>
Pixel 5: <7>
Pixel 6: <7>
Pixel 7: <6>
Pixel 8: <6>

First, we read Pixel 0 into K. _K is then simply <7>, since _ is the empty string at this point. Is the string <7> already in the string table? Of course, since all single character strings were put in the table by InitializeStringTable(). So set _ equal to <7>, and go to the top of the loop.

Read Pixel 1 into K. Does _K (<7><7>) exist in the string table? No, so we get to do some real work. We write the code associated with _ to output (write <7> to output), and add _K (<7><7>) to the table as entry 258. Store K (<7>) into _. Note that although we have added the string consisting of Pixel 0 and Pixel 1 to the table, we _re-use_ Pixel 1 as the beginning of the next string.

Back at the top of the loop. We read Pixel 2 into K. Does _K (<7><7>) exist in the string table? Yes, the entry we just added, entry 258, contains exactly <7><7>. So we just add K onto the end of _, so that _ is now <7><7>.

Back at the top of the loop. We read Pixel 3 into K. Does _K (<7><7><8>) exist in the string table? No, so write the code associated with _ (<258>) to output, and add _K to the table as entry 259. Store K (<8>) into _.

Back at the top of the loop. We read Pixel 4 into K. Does _K (<8><8>) exist in the string table? No, so write the code

TIFF 5.0 page 45

associated with _ (<8>) to output, and add _K to the table as entry 260. Store K (<8>) into _.

Continuing, we get the following results:

After reading:	We write to output:	And add table entry:
Pixel 0		
Pixel 1	<7>	258: <7><7>
Pixel 2		
Pixel 3	<258>	259: <7><7><8>
Pixel 4	<8>	260: <8><8>
Pixel 5	<8>	261: <8><7>
Pixel 6		
Pixel 7	<258>	262: <7><7><6>
Pixel 8	<6>	263: <6><6>

WriteCode() also requires some explanation. The output code stream, <7><258><8><8><258><6>... in our example, should be written using as few bits as possible. When we are just starting out, we can use 9-bit codes, since our new string table entries are greater than 255 but less than 512. But when we add table entry 512, we must switch to 10-bit codes. Likewise, we switch to 11-bit codes at 1024, and 12-bit codes at 2048. We will somewhat arbitrarily limit ourselves to 12-bit codes, so that our table can have at most 4096 entries. If we push it any farther, tables tend to get too large.

What happens if we run out of room in our string table? This is where the afore-mentioned Clear code comes in. As soon as we use entry 4094, we write out a (12-bit) Clear code. (If we wait any longer to write the Clear code, the decompressor might try to interpret the Clear code as a 13-bit code.) At this point, the compressor re-initializes the string table and starts writing out 9-bit codes again.

Note that whenever you write a code and add a table entry, _ is not left empty. It contains exactly one character. Be careful not to lose it when you write an end-of-table Clear code. You

can either write it out as a 12-bit code before writing the Clear code, in which case you will want to do it right after adding table entry 4093, or after the clear code as a 9-bit code. Decompression gives the same result in either case.

To make things a little simpler for the decompressor, we will require that each strip begins with a Clear code, and ends with an EndOfInformation code.

Every LZW-compressed strip must begin on a byte boundary. It need not begin on a word boundary. LZW compression codes are stored into bytes in high-to-low-order fashion, i.e., FillOrder is assumed to be 1. The compressed codes are written as bytes, not words, so that the compressed data will be identical regardless of whether it is an _II_ or _MM_ file.

Note that the LZW string table is a continuously updated history of the strings that have been encountered in the data. It thus reflects the characteristics of the data, providing a high degree of adaptability.

LZW Decoding

The procedure for decompression is a little more complicated, but still not too bad:

```
     while ((Code = GetNextCode()) != EoiCode) {
          if (Code == ClearCode) {
               InitializeTable();
               Code = GetNextCode();
               if (Code == EoiCode)
                    break;
               WriteString(StringFromCode(Code));
               OldCode = Code;
          }  /* end of ClearCode case */

          else {
               if (IsInTable(Code)) {
                    WriteString(StringFromCode(Code));
                    AddStringToTable(StringFromCode(OldCode)+Firs
tChar(StringFromCode(Code)));
                    OldCode = Code;
               } else {
                    OutString   =    StringFromCode(OldCode)    +
FirstChar(StringFromCode(OldCode));
                    WriteString(OutString);
                    AddStringToTable(OutString);
                    OldCode = Code;
               }
          } /* end of not-ClearCode case */
     } /* end of while loop */
```

The function GetNextCode() retrieves the next code from the LZW-coded data. It must keep track of bit boundaries. It knows that the first code that it gets will be a 9-bit code. We add a table entry each time we get a code, so GetNextCode() must switch over to 10-bit codes as soon as string #511 is stored into the table.

The function StringFromCode() gets the string associated with a

particular code from the string table.

The function AddStringToTable() adds a string to the string table. The _+_ sign joining the two parts of the argument to AddStringToTable indicate string concatenation.

StringFromCode() looks up the string associated with a given code.

WriteString() adds a string to the output stream.

TIFF 5.0 page 47

When SamplesPerPixel Is Greater Than 1

We have so far described the compression scheme as if SamplesPerPixel were always 1, as will be be the case with palette color and grayscale images. But what do we do with RGB image data?

Tests on our sample images indicate that the LZW compression ratio is nearly identical regardless of whether PlanarConfiguration=1 or PlanarConfiguration=2, for RGB images. So use whichever configuration you prefer, and simply compress the bytes in the strip.

It is worth cautioning that compression ratios on our test RGB images were disappointing low: somewhere between 1.1 to 1 and 1.5 to 1, depending on the image. Vendors are urged to do what they can to remove as much noise from their images as possible. Preliminary tests indicate that significantly better compression ratios are possible with less noisy images. Even something as simple as zeroing out one or two least-significant bitplanes may be quite effective, with little or no perceptible image degradation.

Implementation

The exact structure of the string table and the method used to determine if a string is already in the table are probably the most significant design decisions in the implementation of a LZW compressor and decompressor. Hashing has been suggested as a useful technique for the compressor. We have chosen a tree based approach, with good results. The decompressor is actually more straightforward, as well as faster, since no search is involved_strings can be accessed directly by code value.

Performance

Many people do not realize that the performance of any compression scheme depends greatly on the type of data to which it is applied. A scheme that works well on one data set may do poorly on the next.

But since we do not want to burden the world with too many compression schemes, an adaptive scheme such as LZW that performs quite well on a wide range of images is very desirable. LZW may not always give optimal compression ratios, but its adaptive nature and relative simplicity seem to make it a good choice.

Experiments thus far indicate that we can expect compression ratios of between 1.5 and 3.0 to 1 from LZW, with no loss of data, on continuous tone grayscale scanned images. If we zero

TIFF 5.0 page 48

the least significant one or two bitplanes of 8-bit data, higher ratios can be achieved. These bitplanes often consist chiefly of noise, in which case little or no loss in image quality will be perceived. Palette color images created in a paint program generally compress much better than continuous tone scanned images, since paint images tend to be more repetitive. It is not unusual to achieve compression ratios of 10 to 1 or better when using LZW on palette color paint images.

By way of comparison, PackBits, used in TIFF for black and white bilevel images, does not do well on color paint images, much less continuous tone grayscale and color images. 1.2 to 1 seemed to be about average for 4-bit images, and 8-bit images are worse.

It has been suggested that the CCITT 1D scheme could be used for continuous tone images, by compressing each bitplane separately. No doubt some compression could be achieved, but it seems unlikely that a scheme based on a fixed table that is optimized for short black runs separated by longer white runs would be a very good choice on any of the bitplanes. It would do quite well on the high-order bitplanes (but so would a simpler scheme like PackBits), and would do quite poorly on the low-order bitplanes. We believe that the compression ratios would generally not be very impressive, and the process would in addition be quite slow. Splitting the pixels into bitplanes and putting them back together is somewhat expensive, and the coding is also fairly slow when implemented in software.

Another approach that has been suggested uses uses a 2D differencing step following by coding the differences using a fixed table of variable-length codes. This type of scheme works quite well on many 8-bit grayscale images, and is probably simpler to implement than LZW. But it has a number of disadvantages when used on a wide variety of images. First, it is not adaptive. This makes a big difference when compressing data such as 8-bit images that have been _sharpened_ using one of the standard techniques. Such images tend to get larger instead of smaller when compressed. Another disadvantage of these schemes is that they do not do well with a wide range of bit depths. The built-in code table has to be optimized for a particular bit depth in order to be effective.

Finally, we should mention _lossy_ compression schemes. Extensive research has been done in the area of lossy, or non-information-preserving image compression. These techniques generally yield much higher compression ratios than can be achieved by fully-reversible, information-preserving image compression techniques such as PackBits and LZW. Some disadvantages: many of the lossy techniques are so computationally expensive that hardware assists are required. Others are so complicated that most microcomputer software vendors could not afford either the expense of implementation or the increase in application object code size. Yet others

TIFF 5.0 page 49

sacrifice enough image quality to make them unsuitable for publishing use.

In spite of these difficulties, we believe that there will one day be a standardized lossy compression scheme for full color images that will be usable for publishing applications on microcomputers. An International Standards Organization group, ISO/IEC/JTC1/SC2/WG8, in cooperation with CCITT Study Group VIII, is hard at work on a scheme that might be appropriate. We expect that a future revision of TIFF will incorporate this scheme once it is finalized, if it turns out to satisfy the needs of desktop publishers and others in the microcomputer community. This will augment, not replace, LZW as an approved TIFF compression scheme. LZW will very likely remain the scheme of choice for Palette color images, and perhaps 4-bit grayscale images, and may well overtake CCITT 1D and PackBits for bilevel images.

Future LZW Extensions

Some images compress better using LZW coding if they are first subjected to a process wherein each pixel value is replaced by the difference between the pixel and the preceding pixel. Performing this differencing in two dimensions helps some images even more. However, many images do not compress better with this extra preprocessing, and for a significant number of images, the compression ratio is actually worse. We are therefore not making differencing an integral part of the TIFF LZW compression scheme.

However, it is possible that a _prediction_ stage like differencing may exist which is effective over a broad range of images. If such a scheme is found, it may be incorporated in the next major TIFF revision. If so, a new value will be defined for the new _Predictor_ TIFF tag. Therefore, all TIFF readers that read LZW files must pay attention to the Predictor tag. If it is 1, which is the default case, LZW decompression may proceed safely. If it is not 1, and the reader does not recognize the specified prediction scheme, the reader should give up.

Acknowledgements

The original LZW reference has already been given. The use of ClearCode as a technique to handle overflow was borrowed from the compression scheme used by the Graphics Interchange Format (GIF), a small-color-paint-image-file format used by CompuServe that also is an adaptation of the LZW technique. Joff Morgan and Eric Robinson of Aldus were each instrumental in their own way in getting LZW off the ground.

TIFF 5.0 page 50

Appendix G: TIFF Classes

Rationale

TIFF was designed to make life easier for scanner vendors, desktop publishing software developers, and users of these two classes of products, by reducing the proliferation of proprietary scanned image formats. It has succeeded far beyond our expectations in this respect. But we had expected that TIFF would be of interest to only a dozen or so scanner vendors (there weren_t any more than that in 1985), and another dozen or so desktop publishing software vendors. This turned out to be a gross underestimate. The only problem with this sort of success is that TIFF was designed to be powerful and flexible, at the expense of simplicity. It takes a fair amount of effort to handle all the options currently defined in this specification (probably no application does a complete job), and that is currently the only way you can be sure that you will be able to import any TIFF image, since there are so many image-generating applications out there now.

So here is an attempt to channel some of the flexibility of TIFF into more restrictive paths, using what we have learned in the past two years about which options are the most useful. Such an undertaking is of course filled with fairly arbitrary decisions. But the result is that writers can more easily write files that will be successfully read by a wide variety of applications, and readers can know when they can stop adding TIFF features.

The price we pay for TIFF Classes is some loss in the ability to adapt. Once we establish the requirements for a TIFF Class, we can never add new requirements, since old software would not know about these new requirements. (The best we can do at that point is establish new TIFF Classes. But the problem with that is that we could quickly have too many TIFF Classes.) So we must believe that we know what we are doing in establishing these Classes. If we do not, any mistakes will be expensive.

Overview

Four TIFF Classes have been defined:

- Class B for bilevel (1-bit) images
- Class G for grayscale images
- Class P for palette color images
- Class R for RGB full color images

To save time and space, we will usually say _TIFF B_, _TIFF G_, _TIFF P,_ and _TIFF R._ If we are talking about all four types, we may write _TIFF X._

TIFF 5.0 page 51

(Note to fax people: if you are interested in a fax TIFF F

Class, please get together and decide what should be in TIFF Class F files. Let us know if we can help in any way. When you have decided, send us your results, so that we can include the information here.)

Core Requirements

This section describes requirements that are common to all TIFF Class X images.

General Requirements

The following are required characteristics of all TIFF Class X files.

Where there are options, TIFF X writers can do whichever one they want, though we will often recommend a particular choice, but TIFF X readers must be able to handle all of them. Please pay close attention to the recommendations. It is possible that at some point in the future, new and even-simpler TIFF classes will be defined that include only recommended features.

You will need to read at least the first three sections of the main specification in order to fully understand the following discussion.

Defaults. TIFF X writers may, but are not required, to write out a field that has a default value, if the default value is the one desired. TIFF X readers must be prepared to handle either situation.

Other fields. TIFF X readers must be prepared to encounter fields other than the required fields in TIFF X files. TIFF X writers are allowed to write fields such as Make, Model, DateTime, and so on, and TIFF X readers can certainly make use of such fields if they exist. TIFF X readers must not, however, refuse to read the file if such optional fields do not exist.

_MM_ and ëIIí byte order. TIFF X readers must be able to handle both byte orders. TIFF writers can do whichever is most convenient or efficient. Images are crossing the IBM PC/Macintosh boundary (and others as well) with a surprisingly high frequency. We could force writers to all use the same byte order, but preliminary evidence indicates that this will cause problems when we start seeing greater-than-8-bit images. Reversing bytes while scanning could well slow down the scanning process enough to cause the scanning mechanism to stop, which tends to create image quality problems.

Multiple subfiles. TIFF X readers must be prepared for multiple images (i.e., subfiles) per TIFF file, although they are not required to do anything with any image after the first one. TIFF

TIFF 5.0 page 52

X writers must be sure to write a long word of 0 after the last IFD (this is the standard way of signalling that this IFD was the last one) as indicated in the TIFF structure discussion.

If a TIFF X writer writes multiple subfiles, the first one must be the full resolution image. Subsequent subimages, such as reduced resolution images and transparency masks, may be in any

order in the TIFF file. If a reader wants to make use of such subimages, it will have to scan the IFDís before deciding how to proceed.

TIFF X Editors. Editors, applications that modify TIFF files, have a few additional requirements.

TIFF editors must be especially careful about subfiles. If a TIFF editor edits a full-resolution subfile, but does not update an accompanying reduced-resolution subfile, a reader that uses the reduced-resolution subfile for screen display will display the wrong thing. So TIFF editors must either create a new reduced-resolution subfile when they alter a full-resolution subfile, or else they must simply delete any subfiles that they aren_t prepared to deal with.

A similar situation arises with the fields themselves. A TIFF X editor need only worry about the TIFF X required fields. In particular, it is unnecessary, and probably dangerous, for an editor to copy fields that it does not understand. It may have altered the file in a way that is incompatible with the unknown fields.

Required Fields

NewSubfileType. LONG. Recommended but not required.

ImageWidth. SHORT or LONG. (That is, both _SHORT_ and _LONG_ TIFF data types are allowed, and must be handled properly by readers. TIFF writers can use either.) TIFF X readers are not required to read arbitrarily large files however. Some readers will give up if the entire image cannot fit in available memory. (In such cases the reader should inform the user of the nature of the problem.) Others will probably not be able to handle ImageWidth greater than 65535. Recommendation: use LONG, since resolutions seem to keep going up.

ImageLength. SHORT or LONG. Recommendation: use LONG.

RowsPerStrip. SHORT or LONG. Readers must be able to handle any value between 1 and 2**32-1. However, some readers may try to read an entire strip into memory at one time, so that if the entire image is one strip, the application may run out of memory. Recommendation 1: Set RowsPerStrip such that the size of each strip is about 8K bytes. Do this even for uncompressed data, since it is easy for a writer and makes things simpler for

TIFF 5.0 page 53

readers. (Note: extremely wide, high-resolution images may have rows larger than 8K bytes; in this case, RowsPerStrip should be 1, and the strip will just have to be larger than 8K. Recommendation 2: use LONG.

StripOffsets. SHORT or LONG. As explained in the main part of the specification, the number of StripOffsets depends on RowsPerStrip and ImageLength. Recommendation: always use LONG. (LONG must, of course, be used if the file is more than 64K bytes in length.)

StripByteCounts. SHORT or LONG. Many existing TIFF images do not contain StripByteCounts, because, in a strict sense, they are

unnecessary. It is possible to write an efficient TIFF reader that does not need to know in advance the exact size of a compressed strip. But it does make things considerably more complicated, so we will require StripByteCounts in TIFF X files. Recommendation: use SHORT, since strips are not supposed to be very large.

XResolution, YResolution. RATIONAL. Note that the X and Y resolutions may be unequal. A TIFF X reader must be able to handle this case. TIFF X pixel-editors will typically not care about the resolution, but applications such as page layout programs will.

ResolutionUnit. SHORT. TIFF X readers must be prepared to handle all three values for ResolutionUnit.

TIFF Class B - Bilevel

Required (in addition to the above core requirements)

The following fields and values are required for TIFF B files, in addition to the fields required for all TIFF X images (see above).

SamplesPerPixel = 1. SHORT. (Since this is the default, the field need not be present. The same thing holds for other required TIFF X fields that have defaults.)

BitsPerSample = 1. SHORT.

Compression = 1, 2 (CCITT 1D), or 32773 (PackBits). SHORT. TIFF B readers must handle all three. Recommendation: use PackBits. It is simple, effective, fast, and has a good worst-case behavior. CCITT 1D is definitely more effective in some situations, such as scanning a page of body text, but is tough to implement and test, fairly slow, and has a poor worst-case behavior. Besides, scanning a page of 12 point text is not very useful for publishing applications, unless the image data is turned into ASCII text via OCR software, which is outside the scope of TIFF.

PhotometricInterpretation = 0 or 1. SHORT.
A Sample TIFF B Image

```
Offset         Value
(hex)     Name (mostly hex)

Header:
0000 Byte Order     4D4D
0002 Version   002A
0004 1st IFD pointer     00000014

IFD:
0014 Entry Count    000D
0016 NewSubfileType 00FE 0004 00000001  00000000
0022 ImageWidth     0100 0004 00000001  000007D0
002E ImageLength    0101 0004 00000001  00000BB8
003A Compression    0103 0003 00000001  8005 0000
0046 PhotometricInterpretation     0106 0003 00000001  0001 0000
```

```
0052 StripOffsets   0111 0004 000000BC  000000B6
005E RowsPerStrip   0116 0004 00000001  00000010
006A StripByteCounts     0117 0003 000000BC  000003A6
0076 XResolution    011A 0005 00000001  00000696
0082 YResolution    011B 0005 00000001  0000069E
008E Software  0131 0002 0000000E  000006A6
009A DateTime  0132 0002 00000014  000006B6
00A6 Next IFD pointer    00000000
```

Fields pointed to by the tags:

```
00B6 StripOffsets   Offset0, Offset1, ... Offset187
03A6 StripByteCounts     Count0, Count1, ... Count187
0696 XResolution    0000012C 00000001
069E YResolution    0000012C 00000001
06A6 Software  "PageMaker 3.0"
06B6 DateTime  "1988:02:18 13:59:59"
```

Image Data:

```
00000700  Compressed data for strip 10
xxxxxxxx  Compressed data for strip 179
xxxxxxxx  Compressed data for strip 53
xxxxxxxx  Compressed data for strip 160
.
.
.
```

End of example

Comments on the TIFF B example

1. The IFD in our example starts at position hex 14. It could have been anywhere in the file as long as the position is even and greater than or equal to 8, since the TIFF header is 8 bytes long and must be the first thing in a TIFF file.

TIFF 5.0 page 55

2. With 16 rows per strip, we have 188 strips in all.

3. The example uses a number of optional fields, such as DateTime. TIFF X readers must safely skip over these fields if they do not want to use the information. And TIFF X readers must not require that such fields be present.

4. Just for fun, our example has highly fragmented image data; the strips of our image are not even in sequential order. The point is that strip offsets must not be ignored. Never assume that strip N+1 follows strip N. Incidentally, there is no requirement that the image data follows the IFD information. Just the follow the pointers, whether they be IFD pointers, field pointers, or Strip Offsets.

TIFF Class G - Grayscale

Required (in addition to the above core requirements)

SamplesPerPixel = 1. SHORT.

BitsPerSample = 4, 8. SHORT. There seems to be little

justification for working with grayscale images shallower than 4 bits, and 5-bit , 6-bit, and 7-bit images can easily be stored as 8-bit images, as long as you can compress the _unused_ bit planes without penalty. And we can do just that with LZW (Compression = 5.)

Compression = 1 or 5 (LZW). SHORT. Recommendation: use 5, since LZW decompression is turning out to be quite fast.

PhotometricInterpretation = 0 or 1. SHORT. Recommendation: use 1, due to popular user interfaces for adjusting brightness and contrast.

TIFF Class P - Palette Color

Required (in addition to the above core requirements)

SamplesPerPixel = 1. SHORT. We use each pixel value as an index into all three color tables in ColorMap.

BitsPerSample = 1,2,3,4,5,6,7, or 8. SHORT. 1,2,3,4, and 8 are probably the most common, but as long as we are doing that, the rest come pretty much for free.

Compression = 1 or 5. SHORT.

PhotometricInterpretation = 3 (Palette Color). SHORT.

TIFF 5.0 page 56

ColorMap. SHORT.

Note that bilevel and grayscale images can be represented as special cases of palette color images. As soon as enough major applications support palette color images, we may want to start getting rid of distinctions between bilevel, grayscale, and palette color images.

TIFF Class R - RGB Full Color

Required (in addition to the above Core Requirements)

SamplesPerPixel = 3. SHORT. One sample each for Red, Green, and Blue.

BitsPerSample = 8,8,8. SHORT. Shallower samples can easily be stored as 8-bit samples with no penalty if the data is compressed with LZW. And evidence to date indicates that images deeper than 8 bits per sample are not worth the extra work, even in the most demanding publishing applications.

PlanarConfiguration = 1 or 2. SHORT. Recommendation: use 1.

Compression = 1 or 5. SHORT.

PhotometricInterpretation = 2 (RGB). SHORT.

Recommended

Recommended for TIFF Class R, but not required, are the new (as of Revision 5.0) colorimetric information tags. See Appendix H.

Conformance and User Interface

Applications that write valid TIFF X files should include _TIFF B_ and/or _TIFF G_ and/or _TIFF P_ and/or _TIFF R_ and/or in their product spec sheets, if they can write the respective TIFF Class X files. If your application writes all four of these types, you may wish to write it as _TIFF B,G,P,R._ Of course, a term like _TIFF B,_ while fine for communicating with other vendors, will not convey much information to a typical user. In this case, a phrase such as _Standard TIFF Black-and-White Scanned Images_ might be better.

The same terminology guidelines apply to applications that read TIFF Class X files.

If your application reads more kinds of files than it writes, or vice versa, it would be a good idea to make that clear to the buyer. For example, if your application reads TIFF B and TIFF G

TIFF 5.0 page 57

files, but writes only TIFF G files, you should write it that way in the spec sheet.

TIFF 5.0 page 58

Appendix H: Image Colorimetry Information

Chris Sears
210 Lake Street
San Francisco, CA 94118

June 4, 1988
Revised August 8, 1988

I. Introduction

Our goal is to accurately reproduce a color image using different devices. Accuracy requires techniques of measurement and a standard of comparison. Different devices imply device independence. Colorimetry provides the framework to solve these problems. When an image has a complete colorimetric description, in principle it can be reproduced identically on different monitors and using different media, such as offset lithography.

The colorimetry data is specified when the image is created or changed. A scanned image has colorimetry data derived from the filters and light sources of the scanner and a synthetic image has colorimetry data corresponding to the monitor used to create it or the monitor model of the rendering environment. This data is used to map an input image to the markings or colors of a particular output device.

Section II describes various standards organizations and their work in color.
Section III describes our motivation for seeking these tags.
Section IV describes our goals of reproduction.
Sections V, VI and VII introduce the colorimetry tags.
Section VIII specifies the tags themselves.
Section IX describes the defaults.
Section X discusses the limitations and some of the other issues.
Section XI provides a few references.

II. Related Standards

TIFF is a general standard for describing image data. It would be foolish for us to change TIFF in a way that did not match

existing industry and international standards. Therefore, we have taken pains to note in the discussion below the efforts of various standards organizations and select defaults from the work of these organizations.

CIE (Commission Internationale de líEclairage) The basis of the colorimetry information is the CIE 1931 Standard Observer [3]. While other color models could be supported [1] [4], CIE 1931 XYZ is the international standard accepted across industries for specifying color and CIE xyY is the chromaticity diagram associated with CIE 1931 XYZ tristimulus values.

TIFF 5.0 page 59

NTSC (National Television System Committee) NTSC is of interest primarily for historical reasons and its use in encoding television data. Manufacturing standards for monitors have for some time drifted significantly from the 1953 NTSC colorimetry specification.

SMPTE (Society of Motion Picture and Television Engineers) Most of the work by NTSC has been largely subsumed by SMPTE. This organization has a set of standards called "Recommended Practices" that apply to various technical aspects of film and television production [5] [6].

ISO (International Standards Organization) ISO has become involved in color standards through work on a color addendum to "Office Document Architecture (ODA) and Interchange Format" [7].

ANSI (American National Standards Institute) ANSI is the American representative to ISO .

III. Motivation

Our motivation for defining these tags stems from our research and development in color separation technology. With the information described here and the RGB pixel data, we have all of the information necessary for generating high-quality color separations. We could supply the colorimetry information outside of the image file. But since TIFF provides a convenient mechanism for bundling all of the relevant information in a single place, tags are defined to describe this information in color TIFF files.

A color image rendered with incorrect colorimetry information looks different from the original. One of our early test images has an artifact in it where the image was scanned with one set of primaries and color ramps were overlaid on top of it with different primaries. The blue ramp looked purple when we printed it. Using incorrect gamma tables or white points can also lead to distorted images. The best way to avoid these kinds of errors is to allow the creator of an image to supply the colorimetry information along with the RGB values [1] [2].

The purpose of the colorimetry data is to allow a projective transformation from the primaries and white point of the image to the primaries and white point of the rendering media. Gamma reflects the non-linear transfer gradient of real media.

IV. Colorimetric Color Reproduction

Earlier we said that given the proper colorimetric data an image could be rendered identically using two different calibrated devices. By identical, we mean colorimetric reproduction [9].

TIFF 5.0 page 60

Specifically, the chromaticities match and the luminance is scaled to correspond to the luminance range of the output device. Because of this, we only need the chromaticity coordinates of the white point and primaries. The absolute luminance is arbitrary and unnecessary.

V. White Point

In TIFF 4.0, the white point was specified as D65. This appendix allocates a separate tag for describing the white point and D65 is the logical default since it is the SMPTE standard [6].

The white point is defined colorimetrically in the CIE xyY chromaticity diagram. While it is rare for monitors to differ from D65, scanned images often have different white points. Rendered images can have arbitrary white points. The graphic arts use D50 as the standard viewing light source [8].

VI. Primary Chromaticities

In TIFF 4.0, the primary color chromaticities matched the NTSC specification. With the wide variety of color scanners, monitors and renderers, TIFF needs a mechanism for accurately describing the chromaticities of the primary colors. We use SMPTE as the default chromaticity since conventional monitors are closer to SMPTE and some monitors (Conrac 6545) are manufactured to the SMPTE specifications. We doními use the NTSC chromaticities and white point because present day monitors doními use them and must be _matrixed_ to approximate them.

As an example, the primary color chromaticities used by the Sony Trinatron differ from those recommended by SMPTE. In general, since real monitors vary from the industry standards, the chromaticities of primaries are described in the CIE xyY system. This allows a reproduction system to compensate for the differences.

VII. Color Response Curves

This tag defines three color response curves, one each for red, green, and blue color information. The width of each entry is 16 bits, as implied by the type SHORT. The minimum intensity is represented by 0 and the maximum by 65535. For example, black is represented by 0,0,0 and white by 65535, 65535, 65535. The length of each curve is 2**BitsPerSample. A ColorResponseCurves field for RGB data where each of the samples is 8 bits deep would have 3*256 entries. The 256 red entries would come first, followed by 256 green entries, followed by 256 blue entries.

The purpose of the ColorResponseCurves field is to act as a lookup table mapping sample values to specific intensity values,

TIFF 5.0 page 61

so that an image created on one system can be displayed on another with minimal loss of color fidelity. The ColorResponseCurves field thus describes the _gamma_ of an image, so that a TIFF reader on another system can compensate for both the image gamma and the gamma of the reading system.

Gamma is a term that relates to the typically nonlinear response of most display devices, including monitors. In most display systems, the voltage applied to the CRT is directly proportional to the value of the red, green, or blue sample. However, the resulting luminance emitted by the phosphor is not directly proportional to the voltage. This relationship is approximated in most displays by

```
luminance = voltage ** gamma
```

The NTSC standard gamma of 2.2 adequately describes most common video systems. The standard gamma of 2.2 implies a dim viewing surround. (We know of no SMPTE recommended practice for gamma.) The following example uses an 8 bit sample with value of 127.

```
voltage = 127 / 255 = 0.4980
luminance = 0.4980 ** 2.2 = 0.2157
```

In the examples below, we only consider a single primary and therefore only a single curve. The same analysis applies to each of the red, green, and blue primaries and curves. Also, and without loss of generality, we assume that there is no hardware color map, so that we must alter the pixel values themselves. If there is a color map, the manipulations can be done on the map instead of on the pixels.

If no ColorResponseCurves field exists in a color image, the reader should assume a gamma of 2.2 for each of the primaries. This default curve can be generated with the following C code:

```
     ValuesPerSample = 1 << BitsPerSample;
     for (curve[0] = 0, i = 1; i < ValuesPerSample; i++)
          curve[i] =  floor (pow  (i /  (ValuesPerSample -  1.0),
2.2) * 65535.0 + .5);
```

The displaying or rendering application can know its own gamma, which we will call the _destination gamma._ (An uncalibrated system can usually assume that its gamma is 2.2 without going too far wrong.) Using this information the application can compensate for the gamma of the image, as we shall see below.

If the source and destination systems are both adequately described by a gamma of 2.2, the writer would omit the ColorResponseCurves field, and the reader can simply read the image directly into the frame buffer. If a writer writes out the ColorResponseCurves field, then a reader must assume that the gammas differ. A reader must then perform the following computation on each sample in the image:

TIFF 5.0 page 62

```
     NewSampleValue =   floor  (pow   (curve[OldSampleValue]   /
65535.0, 1.0 / DestinationGamma) *
       (ValuesPerSample - 1.0) + .5);
```

Of course, if the _gamma_ of the destination system is not well-approximated with an exponential function, an arbitrary table lookup may be used in place of raising the value to 1.0 / DestinationGamma.

Leave out ColorResponseCurves if using the default gamma. This saves about 1.5K in the most common case, and, after all, omission is the better part of compression.

Do not use this field to store frame buffer color maps. Use instead the ColorMap field. Note, however, that ColorResponseCurves may be used to refine the information in a ColorMap if desired.

The above examples assume that a single parameter gamma system adequately approximates the response characteristics of the image source and destination systems. This will usually be true, but our use of a table instead of a single gamma parameter gives the flexibility to describe more complex relationships, without requiring additional computation or complexity.

VIII. New Tags and Changes
The following tags should be placed in the "Basic Fields" section of
the TIFF specification:

White Point
Tag = 318 (13E)
Type = RATIONAL
N = 2

The white point of the image. Note that this value is described using the 1931 CIE xyY chromaticity diagram and only the chromaticity is specified. The luminance component is arbitrary and not specified. This can correspond to the white point of a monitor that the image was painted on, the filter set/light source combination of a scanner, or to the white point of the illumination model of a rendering package.

Default is the SMPTE white point, D65: x = 0.313, y = 0.329.

The ordering is x, y.

PrimaryChromaticities
Tag = 319 (13F)
Type = RATIONAL
N = 6

TIFF 5.0 page 63

The primary color chromaticities. Note that these values are described using the 1931 CIE xyY chromaticity diagram and only the chromaticities are specified. For paint images, these represent the chromaticities of the monitor and for scanned images they are derived from the filter set/light source combination of a scanner.

Default is the SMPTE primary color chromaticities:

Red: x = 0.635 y = 0.340
Green: x = 0.305 y = 0.595
Blue: x = 0.155 y = 0.070

The ordering is red x, red y, green x, green y, blue x, blue y.

Color Response Curves

Default for ColorResponseCurves represents curves corresponding to the NTSC standard gamma of 2.2.

IX. Defaults

The defaults used by TIFF reflect industry standards. Both the WhitePoint and PrimaryChromaticities tags have defaults that are promoted by SMPTE . In addition, the default for the ColorResponseCurves tag matches the NTSC specification of a gamma of 2.2.

The purpose of these defaults is to allow reasonable results in the absence of accurate colorimetry data. An uncalibrated scanner or paint system produces an image that be displayed identically, though probably incorrectly on two different but calibrated systems. This is better then the uncertain situation where the image might be rendered differently on two different but calibrated systems.

X. Limitations and Issues

This section discusses several of the limitations and issues involved in colorimetric reproduction.

Scope of Usefulness

For many purposes the data recommended here is unnecessary and can be omitted. For presentation graphics where there are only a few colors, being able to tell red from green is probably good enough. In this case the tags can be ignored and there is no overhead. In more demanding color reproduction environments, this data allows images to be described device independently and at small cost.

TIFF 5.0 page 64

User Burdens

The data we recommend isnít a user burden; it is really a systems issue. It allows a systems solution but doesnít require user intercession. Calibration however is a separate issue. It is likely to involve the user.

Resolution Versus Fidelity

Some manufacturers supply greater than 24 bits of resolution for color specification. The purpose of this is either to avoid artifacts such as contouring in the shadows or in some cases to be more specific or device independent about the color. Both reasons can be misguided. Other, less expensive techniques can be used to prevent artifacts, such as deeper color maps. As for accuracy, fidelity is more important than precision.

Colorimetric Color Reproduction

There are other choices for objectives of color reproduction [9]. Spectral color reproduction is a stronger condition and most are weaker, such as preferred color reproduction. While device independent spectral color reproduction is impossible, device independent colorimetric reproduction is possible, within a tolerance and within the limits of the gamuts of the devices. By choosing a strong criteria we allow the important objectives of weaker criteria, such as preferred color reproduction, to be part of design packages.

Metamerism

If two patches of color are identical under one light and different under another, they are said to be metameric pairs. Colorimetric color reproduction is a weaker condition than spectral color reproduction and hence allows metamerism problems. By standardizing the viewing conditions we can largely finesse the metamerism problem for print. Because television is self-luminous and doesnít use spectral absorption, metamerism isnít so much a problem.

Color Models - xyY Versus Luv, etc.

We choose xyY over Luv [1] because XYZ is the international standard for color specification and xyY is the chromaticity diagram associated with XYZ. Luv is meant for color difference measurement.

Ambient Environment And Viewing Surrounds

The viewing environment affects how the eye perceives color. The eye adapts to a dark room and it adapts to a colored surround. While these problems can be compensated for within the colorimetric framework [4], it is much better to finesse them by standardizing. The design environment should match the intended

TIFF 5.0 page 65

viewing environment. Specifically it should not be a pitch dark room and, on average, it should be of a neutral color. For print, ANSI recommends a Munsell N-8 surface [8].

XI. References

In particular, we would like to mention the work of Stuart Ring of the Copy Products Division of the Eastman Kodak Company. He and his colleagues are promoting a color data interchange paradigm. They are working closely with the ISO 8613 Working Group [7].

[1] Color Data Interchange Paradigm, Eastman Kodak, Rochester, New York, 7 December 1987.

[2] Color Reproduction and Illumination Models, Roy Hall, International Summer Institute: State of the Art in Computer Graphics, 1986.

[3] CIE Colorimetry: Official Recommendations of the International Commission on Illumination, Publication 15-2, 1986.

[4] Color Science: Concepts and Methods, Quantitative Data and Formulae, Gunter Wyszecki, W.S. Stiles, John Wiley and Sons, Inc., New York, New York, 1982.

[5] Color Monitor Colorimetry, SMPTE Recommended Practice RP 145-1987.

[6] Color Temperature for Color Television Studio Monitors, SMPTE Recommended Practice RP 37.

[7] Office Document Architecture (ODA) and Interchange Format_Addendum on Colour, ISO 8613 Working Draft.

[8] ANSI Standard PH 2.30-1985.

[9] The Reproduction of Colour in Photography, Printing and Television, R. W. G. Hunt, Fountain Press, Tolworth, England, 1987.

[10] Raster Graphics Handbook, The Conrac Corporation, Van Nostrand Reinhold Company, New York, New York, 1985. Good description of gamma.

TIFF 5.0 page 66

Appendix I: Horizontal Differencing Predictor

This appendix, written after the release of Revision 5.0 of the TIFF specification, is still in draft form. Please send any comments to the Aldus Developers Desk.

Revision 5.0 of the TIFF specification defined a new tag called _Predictor_ that describes techniques that may be used in conjuction with TIFF compression schemes. We now define a Predictor that greatly improves compression ratios for some images.

The horizontal differencing predictor is assigned the tag value Predictor = 2:

Predictor

Tag = 317 (13D)
Type = SHORT
N   = 1

A predictor a mathematical operator that is applied to the image data before the encoding scheme is applied. Currently (as of revision 5.0) this tag is used only with LZW (Compression=5) encoding, since LZW is probably the only TIFF encoding scheme that benefits significantly from a predictor step. See Appendix F.

1 = No prediction scheme used before coding.
2 = Horizontal differencing. See Appendix I.

Default is 1.

The algorithm

The idea is to make use of the fact that many continuous tone images rarely vary much in pixel value from one pixel to the next. In such images, if we replace the pixel values by differences between consecutive pixels, many of the differences should be 0, plus or minus 1, and so on. This reduces the apparent information content, and thus allows LZW to encode the data more compactly.

Assuming 8-bit grayscale pixels for the moment, a basic C implementation might look something like this:

```
     char image[ ][ ];
     int  row, col;

     /* take horizontal differences:
      */
     for (row = 0; row < nrows; row++)
```


TIFF 5.0   page 67

```
          for (col = ncols - 1; col >= 1; col--)
               image[row][col] -= image[row][col-1];
```

If we don_t have 8-bit samples, we need to work a little harder, so that we can make better use of the architecture of most CPUs. Suppose we have 4-bit samples, packed two to a byte, in normal TIFF uncompressed (i.e., Compression=1) fashion. In order to find differences, we want to first expand each 4-bit sample into an 8-bit byte, so that we have one sample per byte, low-order justified. We then perform the above horizontal differencing. Once the differencing has been completed, we then repack the 4-bit differences two to a byte, in normal TIFF uncompressed fashion.

If the samples are greater than 8 bits deep, expanding the samples into 16-bit words instead of 8-bit bytes seems like the best way to perform the subtraction on most computers.

Note that we have not lost any data up to this point, nor will we lose any data later on. It might at first seem that our differencing might turn 8-bit samples into 9-bit differences, 4-bit samples into 5-bit differences, and so on. But it turns out that we can completely ignore the _overflow_ bits caused by subtracting a larger number from a smaller number and still

reverse the process without error. Normal twos complement arithmetic does just what we want. Try an example by hand if you need more convincing.

Up to this point we have implicitly assumed that we are compressing bilevel or grayscale images. An additional consideration arises in the case of color images.

If PlanarConfiguration is 2, there is no problem. Differencing proceeds the same way as it would for grayscale data.

If PlanarConfiguration is 1, however, things get a little trickier. If we didnít do anything special, we would be subtracting red sample values from green sample values, green sample values from blue sample values, and blue sample values from red sample values, which would not give the LZW coding stage much redundancy to work with. So we will do our horizontal differences with an offset of SamplesPerPixel (3, in the RGB case). In other words, we will subtract red from red, green from green, and blue from blue. The LZW coding stage is identical to the SamplesPerPixel=1 case. We require that BitsPerSample be the same for all 3 samples.

Results and guidelines

LZW without differencing works well for 1-bit images, 4-bit grayscale images, and synthetic color images. But natural 24-bit color images and some 8-bit grayscale images do much better with differencing. For example, our 24-bit natural test images hardly

TIFF 5.0 page 68

compressed at all using _plain_ LZW: the average compression ratio was 1.04 to 1. The average compression ratio with horizontal differencing was 1.40 to 1. (A compression ratio of 1.40 to 1 means that if the uncompressed image is 1.40MB in size, the compressed version is 1MB in size.)

Although the combination of LZW coding with horizontal differencing does not result in any loss of data, it may be worthwhile in some situations to give up some information by removing as much noise as possible from the image data before doing the differencing, especially with 8-bit samples. The simplest way to get rid of noise is to mask off one or two low-order bits of each 8-bit sample. On our 24-bit test images, LZW with horizontal differencing yielded an average compression ratio of 1.4 to 1. When the low-order bit was masked from each sample, the compression ratio climbed to 1.8 to 1; the compression ratio was 2.4 to 1 when masking two bits, and 3.4 to 1 when masking three bits. Of course, the more you mask, the more you risk losing useful information along with the noise. We encourage you to experiment to find the best compromise for your device. For some applications it may be useful to let the user make the final decision.

Interestingly, most of our RGB images compressed slightly better using PlanarConfiguration=1. One might think that compressing the red, green, and blue difference planes separately (PlanarConfiguration=2) might give better compression results than mixing the differences together before compressing (PlanarConfiguration=1), but this does not appear to be the case.

Incidentally, we tried taking both horizontal and vertical differences, but the extra complexity of two-dimensional differencing did not appear to pay off for most of our test images. About one third of the images compressed slightly better with two-dimensional differencing, about one third compressed slightly worse, and the rest were about the same.

TIFF 5.0 page 69

Appendix J: Palette Color

This appendix, written after the release of Revision 5.0 of the TIFF specification, is still in draft form. Please send any comments to the Aldus Developers Desk.

Revision 5.0 of the TIFF specification defined a new PhotometricInterpretation value called _palette color._ We have been wondering lately if this additional complexity is worth the implementation expense. If not, let_s drop it before too many people start creating palette color images.

The Proposal

Instead of a separate palette color image type, there seems to be no compelling reason why palette (mapped) color images should not be stored as _full color_ (usually 24-bit) RGB images.

Objections

The most obvious objection is the amount of space required. But if you care about how much space the image takes up on disk, you should use LZW compression, which is ideally suited to most palette color images. (LZW compresses most paint-type palette color images 5:1 or more.) And if you use LZW compression, it turns out that palette color images stored as full color images compress to almost exactly the same size as palette color images stored as palette color images. That is, with LZW compression, there is no penalty for storing palette color images as full color RGB images. The resulting file may be a few percent larger, but it will not be three times as large. See Appendix F for more information on how LZW works.

Another objection might be that an application might want to process the image differently if it is _really_ a palette color image. But we can easily add auxiliary information that can help a TIFF reader to quickly categorize color images if it wants to. See the _New tags_ section below.

Benefits

It may be obvious, but it is probably worth discussing why we want to abolish palette color images as a distinct classification.

The main problem is that palette color as a separate type makes life more hazardous and confusing for users. The confusion factor is aggravated because users already have to be somewhat aware of distinctions between bilevel, grayscale, and color

TIFF 5.0 page 70

images. Having two main types of color images is hard for many users to grasp, and it is probably not possible to totally hide this complexity from the user in certain situations. The hazard level goes up because some applications may accept palette color but not full color images, or full color but not palette color images, or may accept 8-bit palette color images but not 4-bit or 3-bit palette color images.

The second problem is that writing and maintaining code to deal with an additional image type is somewhat expensive for TIFF readers. The cost of supporting palette color images will depend on the application, but we believe that, in general, the cost will be substantial. It seems to make more sense to put the burden on TIFF writers to convert palette color images into full color image form than to make TIFF readers handle an additional color image type, since there are more TIFF readers than writers at this point.

New tags

Here are some proposed new tags that can help to classify color images, and make up for not having a separate palette color class. They are not required for TIFF Class R , but are strongly recommended for color TIFF images created by palette-type color paint programs.

ColorImageType
Tag = 318 (13E)
Type = SHORT
N = 1

Gives TIFF color image readers a better idea of what kind of color image it is. There will be borderline cases.

1 = Continuous tone, natural image.
2 = Synthetic image, using a greatly restricted range of colors. Such images are produced by most color paint programs. See ColorList for a list of colors used in this image.

Default is 1.

ColorList
Tag = 319 (13F)
Type = BYTE or SHORT
N   = the number of colors that are used in this image, times SamplesPerPixel

A list of colors that are used in this image. Use of this field is only practical for images containing a greatly restricted (usually less than or equal to 256) range of colors. ColorImageType should be 2. See ColorImageType.

TIFF 5.0 page 71

The list is organized as an array of RGB triplets, with no pad. The RGB triplets are not guaranteed to be in any particular order. Note that the red, green, and blue components can either be a BYTE or a SHORT in length. BYTE should be sufficient for most applications.

No default.

MICROSOFT EXCEL FILE FORMAT

Microsoft Excel is a popular spreadsheet. It uses a file format called BIFF (Binary File Format). There are many types of BIFF records. Each has a 4 byte header. The first two bytes are an opcode that specifies the record type. The second two bytes specify record length. Header values are stored in byte-reversed form (less significant byte first). The rest of the record is the data itself (Figure 2-1).

Figure 2-1. BIFF record header.

```
                 |  Record Header    |  Record Body
Byte Number      |  0    1    2    3 |  0    1   ...
                ----------------------------------
Record Contents  | XX | XX | XX | XX | XX | XX | ...
                ----------------------------------
                | opcode   | length  | data
```

Each X represents a hexadecimal digit
Two X's form a byte. The least significant (low) byte of the opcode is byte 0 and the most significant (high) byte is byte 1. Similarly, the low byte of the record length field is byte 2 and the high byte is byte 3.

BOF (Beginning of File)

The first record in every spreadsheet is always of the BOF type (Figure 2-2).

Figure 2-2. BOF record.

```
           |  Record Header    |    Record Body    |
Byte       |  0    1    2    3 |  0    1    2    3 |
           ----------------------------------------
Contents   | 09 | 00 | 04 | 00 | 02 | 00 | 10 | 00 |
           ----------------------------------------
           | opcode  | length  | version |  file   |
           |         |         |  number |  type   |
```

The first two bytes, arranged with the low byte first, show that the opcode for BOF is 09h. The second two bytes indicate that the record body is 4 bytes long. The first two bytes of the body are the version number (2 for the initial version of Excel). The last two bytes are the file type. Type 10h is a worksheet file.

Relating Spreadsheet Cells to Record Data Bytes

A spreadsheet appears on a screen or printout as a matrix of rectangular cells. Each column is identified by a letter at its top, and each row is identified by a number. Thus cell A1 is in the first column and the first row. Cell C240 is in the third column and the 240th row. This scheme identifies cells in a way easily understood by people. However, it is not particularly convenient for computers, as they do not handle letters efficiently. They are best at dealing with binary numbers. Thus, Excel stores cell identifiers as binary numbers, that people can read as hexadecimal. The first number in the system is 0 rather than 1.

Figure 2-3, which shows the form of an INTEGER record, illustrates the storage of column and row information.

Figure 2-3. INTEGER record.

```
      |  Record Header    |  Record Body
Byte  |  0    1    2    3 |  0    1    2    3    4    5    6    7    8 |
      -----------------------------------------------------------------
Value | 02 | 00 | 09 | 00 | 00 | 00 | 02 | 00 | 00 | 00 | 00 | 39 | 00 |
      -----------------------------------------------------------------
      | opcode  | length  |   row   | column  |    rgbAttr    |    w    |
```

Opcode 2 indicates an integer record. The length bytes show that the record body is 9 bytes long. Row 0 in the body corresponds to spreadsheet row 1. Row 1 corresponds to spreadsheet row 2, and so on. Column 2 corresponds to spreadsheet column C. Thus, Figure 2-3 deals with cell C1. The next three bytes, labeled "rgbAttr," specify cell attributes (Table 2-3). The final pair of bytes, (labeled "w") holds the integer's value. Here it is 39H or 57 decimal. Thus the record specifies that cell C1 of the spreadsheet contains an integer with the value 57.

Standard File Record Order

Excel worksheet files have each record type in a predetermined position. A file need not have all types, but the ones that are present are always be in the same order. Table 2-1 lists the record types for Excel document (spreadsheet) files, in the order they would appear in a BIFF file. Table 2-2 lists the types in opcode order.

Several record types in a BIFF file, namely, ROW, BLANK, INTEGER, NUMBER, LABEL, BOOLERR, FORMULA, and COLUMN DEFAULT, describe the contents of a cell. These records contain a 3 byte attribute field labeled "rgbAttr". The following table describes how the bits in the field correspond to cell attributes.

Table 2-1. Cell Attributes

Byte Offset	Bit	Description	Contents
0	7	Cell is not hidden	0b
		Cell is hidden	1b
	6	Cell is not locked	0b

Byte	Bit	Description	Contents
		Cell is locked	1b
	5-0	Reserved, must be 0	000000b
	7-6	Font number (4 possible)	
	5-0	Cell format code	
2	7	Cell is not shaded	0b
		Cell is shaded	1b
	6	Cell has no bottom border	0b
		Cell has a bottom border	1b
	5	Cell has no top border	0b
		Cell has a top border	1b
	4	Cell has no right border	0b
		Cell has a right border	1b
	3	Cell has no left border	0b
		Cell has a left border	1b
	2-0	Cell alignment code	
		general	000b
		left	001b
		center	010b
		right	011b
		fill	100b
		Multiplan default align.	111b

The font number field is a zero-based index into the document's table of fonts. the cell format code is a zero-based index into the document's table of picture formats. There are 21 different standard formats. Additional custom formats may be defined by the user. See the FONT and FORMAT record descriptions form additonal details.

Table 2-2. Excel Record Type in Order of Appearance

Record Type	Opcode (Hexadecimal)
BOF	09
FILEPASS	2F
INDEX	0B
CALCCOUNT	0C
CALCMODE	0D
PRECISION	0E
REFMODE	0F
DELTA	10
ITERATION	11
1904	22
BACKUP	40
PRINT ROW HEADERS	2A
PRINT GRIDLINES	2B
HORIZONTAL PAGE BREAKS	1B
VERTICAL PAGE BREAKS	1A
DEFAULT ROW HEIGHT	25
FONT	31
FONT2	32
HEADER	14
FOOTER	15
LEFT MARGIN	26
RIGHT MARGIN	27
TOP MARGIN	28
BOTTOM MARGIN	29
COLWIDTH	24
EXTERNCOUNT	16
EXTERNSHEET	17
EXTERNNAME	23
FORMATCOUNT	1F
FORMAT	1E
NAME	18
DIMENSIONS	00
COLUMN DEFAULT	20
ROW	08
BLANK	01
INTEGER	02
NUMBER	03
LABEL	04
BOOLERR	05
FORMULA	06
ARRAY	21
CONTINUE	3C
STRING	07
TABLE	36
TABLE2	37
PROTECT	12

WINDOW PROTECT	19
PASSWORD	13
NOTE	1C
WINDOW1	3D
WINDOW2	3E
PANE	41
SELECTION	1D
EOF	0A

Table 2-3. Excel Record Types in Opcode Order

Record Type	Opcode (hexadecimal)
DIMENSIONS	00
BLANK	01
INTEGER	02
NUMBER	03
LABEL	04
BOOLERR	05
FORMULA	06
STRING	07
ROW	08
BOF	09
EOF	0A
INDEX	0B
CALCCOUNT	0C
CALCMODE	0D
PRECISION	0E
REFMODE	0F
DELTA	10
ITERATION	11
PROTECT	12
PASSWORD	13
HEADER	14
FOOTER	15
EXTERNCOUNT	16
EXTERNSHEET	17
NAME	18
WINDOW PROTECT	19
VERTICAL PAGE BREAKS	1A
HORIZONTAL PAGE BREAKS	1B
NOTE	1C
SELECTION	1D
FORMAT	1E
FORMATCOUNT	1F
COLUMN DEFAULT	20
ARRAY	21
1904	22
EXTERNNAME	23
COLWIDTH	24
DEFAULT ROW HEIGHT	25
LEFT MARGIN	26
RIGHT MARGIN	27
TOP MARGIN	28
BOTTOM MARGIN	29
PRINT ROW HEADERS	2A
PRINT GRIDLINES	2B
FILEPASS	2F
FONT	31
FONT2	32
TABLE	36
TABLE2	37
CONTINUE	3C
WINDOW1	3D
WINDOW2	3E
BACKUP	40
PANE	41

Worksheet Record Types in Opcode Order

The following section lists all record types in opcode order. It gives a specification and byte-by-byte breakdown of each type. Note that Excel terminology refers to spreadsheets or worksheets as "documents."

DIMENSIONS 00h 0d

Record Type: DIMENSIONS
Description: Entire dimensions or range of a spreadsheet

```
Record Body Length: 8 bytes
Record Body Byte Structure:
      Byte Number     Byte Description               Contents (hex)
           0-1        First row
           2-3        Last row plus 1
           4-5        First column
           6-7        Last column plus 1
Note: The last row and column in the record are both one greater than the highest
numbered occupied ones.

BLANK                      01h                          1d

Record Type: BLANK
Description: Cell with no formula or value
Record Body Length: 7 bytes
Record Body Byte Structure:
      Byte Number     Byte Description               Contents (hex)
           0-1        Row
           2-3        Column
           4-6        Cell attributes (rgbAttr) (Table 2-3)

INTEGER                    02h                          2d

Record Type: INTEGER
Description: Constant unsigned integer
Record Body Length: 9 bytes
Record Body Byte Structure:
      Byte Number     Byte Description               Contents (hex)
           0-1        Row
           2-3        Column
           4-6        Cell attributes (rgbAttr) (Table 2-3)
           7-8        Unsigned integer value (w)

NUMBER                     03h                          3d

Record Type: NUMBER
Description: Constant floating point number
Record Body Length: 15 bytes
Record Body Byte Structure:
      Byte Number     Byte Description               Contents (hex)
           0-1        Row
           2-3        Column
           4-6        Cell attributes (rgbAttr) (Table 2-3)
           7-14       Floating point number value (IEEE format, see Appendix A)

LABEL                      04h                          4d

Record Type: LABEL
Description: Constant string
Record Body Length: 8 to 263 bytes
Record Body Byte Structure:
      Byte Number     Byte Description               Contents (hex)
           0-1        Row
           2-3        Column
           4-6        Cell attributes (rgbAttr) (Table 2-3)
           7          Length of string
           8-263      ASCII string, 0 to 255 bytes long

BOOLERR                    05h                          5d

Record Type: BOOLERR
Description: Boolean constant or error value
Record Body Length: 9 bytes
Record Body Byte Structure:
      Byte Number     Byte Description               Contents (hex)
           0-1        Row
           2-3        Column
           4-6        Cell attributes (rgbAttr) (Table 2-3)
           7          Boolean or error value
                                Boolean
                                      true                1
                                      false               0
                                Error
```

		#NULL!	0
		#DIV/0!	7
		#VALUE!	0Fh
		#REF!	17h
		#NAME?	1Dh
		#NUM!	24h
		#N/A	2Ah
8	Specifies Boolean or error		
		Boolean	0
		Error	1

FORMULA 06h 6d

Record Type: FORMULA
Description: Name, size, and contents of a formula cell
Record Body Length: 17-272 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Row	
2-3	Column	
4-6	Cell attributes (rgbAttr) (see Table 2-3)	
7	Current value of formula (IEEE format, see Appendix A)	
15	Recalc flag	
16	Length of parsed expression	
17	Parsed expression	

If a formula must be recalculated whenever it is loaded, the recalc flag (byte 15) must be set. Any nonzero value is a set recalc flag. However, a flag value of 3 indicates that the cell is a part of a matrix, and the entire matrix must be recalculated. Bytes 7 through 14 may contain a number, a Boolean value, an error code, or a string. The following tables apply.

Case 1: Bytes 7 - 14 contain a Boolean value.

Byte Number	Byte Description	Contents (hex)
7	otBool	1
8	Reserved	0
9	Boolean value	
10-12	Reserved	0
13-14	fExprO	FFFFh

Case 2: Bytes 7 - 14 contain an error code.

Byte Number	Byte Description	Contents (hex)
7	otErr	2
8	Reserved	0
9	error code	
10-12	Reserved	0
13-14	fExprO	FFFFh

Case 3: Bytes 7 - 14 contain a string.

Byte Number	Byte Description	Contents (hex)
7	otString	0
8-12	Reserved	0
13-14	fExprO	FFFFh

The string value itself is not stored in the field, but rather in a separate record of the STRING type.

STRING 07h 7d

Record Type: STRING
Description: Value of a string in a formula
Record Body Length: variable
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0	Length of the string	
1-256 (max)	The string itself	

The STRING record appears immediately after the FORMULA record that evaluates to the string, unless the formula is in an array. In that case, the string record immediately follows the ARRAY record.

ROW 08h 8d

Record Type: ROW
Description: Specifies a spreadsheet row
Record Body Length: 16 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Row number	
2-3	First defined column in the row	
4-5	Last defined column in the row plus 1	

Byte Number	Byte Description	Contents (hex)
6-7	Row height	
8-9	RESERVED	0
10	Default cell attributes byte	
	Default attributes	1
	Not default attributes	0
11-12	Offset to cell records for this row	
13-15	Cell attributes (rgbAttr) (see Table 2-3)	

BOF 09h 9d

Record Type: BOF
Description: Beginning of file
Record Body Length: 4 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Version number	
	Excel	2
	Multiplan	3
2-3	Document type	
	worksheet	10h
	chart	20h
	macro sheet	40h

If bit 8 of the version number byte pair is high (mask with 0100h to find out), the BIFF file is a Multiplan document.

EOF 0Ah 10d

Record Type: EOF
Description: End of file
Record Body Length: 0 bytes
The EOF record is the last one in a BIFF file. It always takes the form 0A000000h.

INDEX 0Bh 11d

Record Type: INDEX
Description: Contains pointers to other records in the BIFF file, and defines the range of rows used by the document. It is used to simplify searching a file for a particular cell or name.
Record Body Length: variable
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-3	Absolute file position of first NAME record	
4-5	First row that exists	
6-7	Last row that exists plus 1	
8-on	Array of absolute file positions of the blocks of ROW records.	

The INDEX record is optional. If present, it must immediately follow the FILEPASS record. IF there is no FILEPASS record, it must follow the BOF record.

CALCCOUNT 0Ch 12d

Record Type: CALCCOUNT
Description: Specifies the iteration count
Record Body Length: 2
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Iteration Count	

CALCMODE 0Dh 13d

Record Type: CALCMODE
Description: Specifies the calculation mode
Record Body Length: 2
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Calculation mode	
	Manual	0
	Automatic	1
	Automatic, no tables	-1

PRECISION 0Eh 14d

Record Type: PRECISION
Description: Specifies precision of calculations for document
Record Body Length: 2
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Document precision	

```
                        precision as displayed           0
                        full precision                   1
REFMODE                     0Fh                         15d

Record Type: REFMODE
Description: Specifies location reference mode
Record Body Length: 2
Record Body Byte Structure:
      Byte Number     Byte Description            Contents (hex)
          0-1         Reference mode
                        R1C1 mode                      0
                        A1 mode                        1
DELTA                       10h                          16d

Record Type: DELTA
Description: Maximum change for an iterative model
Record Body Length: 8
Record Body Byte Structure:
      Byte Number           Byte Description      Contents (hex)
          0-7               Maximum change (IEEE format, see Appendix A)
ITERATION                   11h                         17d

Record Type: ITERATION
Description: Specifies whether iteration is on
Record Body Length: 2
Record Body Byte Structure:
      Byte Number           Byte Description         Contents (hex)
          0-1               Iteration flag
                              Iteration off               0
                              Iteration on                1
PROTECT                     12h                         18d

Record Type: PROTECT
Description: Specifies whether the document is protected with a document password
Record Body Length: 2
Record Body Byte Structure:
      Byte Number           Byte Description         Contents (hex)
          0-1               Document protection
                              Not protected               0
                              Protected                   1
PASSWORD                    13h                         19d

Record Type: PASSWORD
Description: Contains encrypted document password
Record Body Length: 2
Record Body Byte Structure:
      Byte Number           Byte Description         Contents (hex)
          0-1               Encrypted password
HEADER                      14h                         20d

Record Type: HEADER
Description: Specifies header string that appears at the top of every page when the
document is printed
Record Body Length: variable
Record Body Byte Structure:
      Byte Number           Byte Description         Contents (hex)
           0                Length of string (bytes)
         1-on               Header string (ASCII)
FOOTER                      15h                         21d

Record Type: FOOTER
Description: Specifies footer string that appears at the bottom of every page when the
document is printed
Record Body Length: variable
Record Body Byte Structure:
      Byte Number           Byte Description         Contents (hex)
          0                 Length of string (bytes)
        1-on                Footer string (ASCII)
EXTERNCOUNT                 16h                         22d

Record Type: EXTERNCOUNT
Description: Specifies the number of documents referenced externally by an Excel
document
```

Record Body Length: 2
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Number of externally referenced documents	

EXTERNSHEET 17h 23d

Record Type: EXTERNSHEET
Description: Specifies a document that is referenced externally by the Excel file. There must be an EXTERNSHEET record for every external file counted by the EXTERNCOUNT record.
Record Body Length: variable
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0	Length of document name	
1-on	Document name	

The document name may be encoded. If so, its first character will be 0, 1 or 2.
0 indicates the document name is an external reference to an empty sheet.
1 indicates the document name has been translated to a less sys-tem-dependent name. This feature is valuable for documents intended for a non-DOS environment.
2 indicates that the externally referenced document is, in fact, the current document.

NAME 18h 24d

Record Type: NAME
Description: User-defined name on the document
Record Body Length: variable
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0	Name attribute Only bits 1 and 2 are ever nonzero. Bit 1 is 1 if the name is a function or command name on a macro sheet. Bit 2 is 1 if the name definition includes: * A function that returns an array, such as TREND or MINVERSE * A ROW or COLUMN function * A user-defined function	
	Name attribute Meaningful only if bit 1 of byte 0 is 1 (the name is a function or command name). Only bits 0 and 1 are ever nonzero. Bit 0 is 1 if the name is a function. Bit 1 is 1 if the name is a command.	
2	Keyboard shortcut. Meaningful only if the name is a command.	
	If no keyboard shortcut	0
	If shortcut exists	ASCII value
3	Length of the name text	
4	Length of the name's definition	
5-?	Text of the name	
?-?	Name's definition (parsed) in internal compressed format	
?	Length of the name's definition (duplicate)	

All NAME records should appear together in a BIFF file.

WINDOW PROTECT 19h 25d

Record Type: WINDOW PROTECT
Description: Specifies whether a document's windows are protected
Record Body Length: 2 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Window protect flag	
	Not protected	0
	Protected	1

VERTICAL PAGE BREAKS 1Ah 26d

```
Record Type: VERTICAL PAGE BREAKS
Description: Lists all column page breaks
Record Body Length: variable
Record Body Byte Structure:
      Byte Number           Byte Description         Contents (hex)
          0-1               Number of page breaks
          2-on              Array containing a 2-byte integer for each
                            column that immediately follows a column page
                            break.  Columns must be sorted in ascending
                            order.
HORIZONTAL PAGE BREAKS      1Bh                         27d

Record Type: HORIZONTAL PAGE BREAKS
Description: Lists all row page breaks
Record Body Length: variable
Record Body Byte Structure:
      Byte Number           Byte Description         Contents (hex)
          0-1               Number of page breaks
          2-on              Array containing a 2-byte integer for each
                            row that immediately follows a row page
                            break.  Rows must be sorted in ascending
                            order.
NOTE                        1Ch                         28d

Record Type: NOTE
Description: Note associated with a cell
Record Body Length: Variable, maximum of 254
Record Body Byte Structure:
      Byte Number           Byte Description         Contents (hex)
          0-1               Row of the note
          2-3               Column of the note
          4-5               Length of the note part of the record
          6-on               Text of the note
Notes longer than 2048 characters must be split among multiple records.  All except the
last one will contain 2048 text characters.  The last one will contain the overflow.
SELECTION                   1Dh                         29d

Record Type: SELECTION
Description: Specifies which cells are selected in a pane of a split window.  It can
also specify selected cells in a window that is not split.
Record Body Length: Variable
Record Body Byte Structure:
      Byte Number           Byte Description         Contents (hex)
           0                Number of pane
                              bottom right                0
                              top right                   1
                              bottom left                 2
                              top left                    3
                              no splits                   3
          1-2               Row number of the active cell
          3-4               Column number of the active cell
          5-6               Reference number of the active cell
          7-8               Number of references in the selection
          9-on              Array of references
Each reference in the array consists of 6 bytes arranged as follows:
      Byte Number           Byte Description
          0-1               First row in the reference
          2-3               Last row in the reference
           4                First column in the reference
           5                Last column in the reference
FORMAT                      1Eh                         30d

Record Type: FORMAT
Description: Describes a picture format in a document.  All FORMAT records must appear
together in a BIFF file.
Record Body Length: Variable
Record Body Byte Structure:
      Byte Number           Byte Description         Contents (hex)
           0                Length of format string
          1-on              Picture format string
FORMATCOUNT                 1Fh                         31d
```

Record Type: FORMATCOUNT
Description: The number of standard FORMAT records in the file. There are 21 different format records.
Record Body Length: 2 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Number of built in format records.	

COLUMN DEFAULT 20h 32d

Record Type: COLUMN DEFAULT
Description: Specifies default cell attributes for cells in a particular column. The default value is overriden for individual cells by a subsequent explicit definition.
Record Body Length: Variable
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Column number of first column for which a default cell is being defined	
2-3	Column number of last column for which a default cell is being defined, plus 1.	
4-on	Array of cell attributes	

ARRAY 21h 33d

Record Type: ARRAY
Description: Describes a formula entered into a range of cells as an array. Occurs immediately after the FORMULA record for the upper left corner of the array.
Record Body Length: variable
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	First row of the array	
2-3	Last row of the array	
4	First column of the array	
5	Last column of the array	
6	Recalculation flag	
	Array is calculated	0
	Needs to be calculated	nonzero
7	Length of parsed expression	
8-on	Parsed expression (array formula)	

1904 22h 34d

Record Type: 1904
Description: Specifies date system used on this spreadsheet
Record Body Length: 2 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Specifies date system used	
	1904 date system	1
	anything else	0

EXTERNNAME 23h 35d

Record Type: EXTERNNAME
Description: An externally referenced name, referring to a work-sheet or macro sheet or to a DDE topic. All EXTERNNAME records associated with a supporting document must directly follow its EXTERNSHEET record.
Record Body Length: Variable
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0	Length of the name	
1-on	The name	

When EXTERNNAME references a DDE topic, Excel may append its most recent values to the EXTERNNAME record. If the record becomes too long to be contained in a single record, it is split into multiple records, with CONTINUE records holding the excess.
COLWIDTH 24h 36d

Record Type: COLWIDTH
Description: Sets column width for a range of columns
Record Body Length: 3 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0	First column in the range	
1	Last column in the range	
2-3	Column width in units of 1/256th of a character	

DEFAULT ROW HEIGHT 25h 37d

Record Type: DEFAULT ROW HEIGHT
Description: Specifies the height of all rows that are not defined explicitly
Record Body Length: 2 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Default row height in units of 1/20th of a point	

LEFT MARGIN	26h	38d

Record Type: LEFT MARGIN
Description: Specifies the left margin in inches when the document is printed
Record Body Length: 8 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-7	Left margin (IEEE format, see Appendix A)	

RIGHT MARGIN	27h	39d

Record Type: RIGHT MARGIN
Description: Specifies the right margin in inches when the document is printed
Record Body Length: 8 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-7	Right margin (IEEE format, see Appendix A)	

TOP MARGIN	28h	40d

Record Type: TOP MARGIN
Description: Specifies the top margin in inches when the document is printed
Record Body Length: 8 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-7	Top margin (IEEE format, see Appendix A)	

BOTTOM MARGIN	29h	41d

Record Type: BOTTOM MARGIN
Description: Specifies the bottom margin in inches when the document is printed
Record Body Length: 8 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-7	Bottom margin (IEEE format, see Appendix A)	

PRINT ROW HEADERS	2Ah	42d

Record Type: PRINT ROW HEADERS
Description: Flag determines whether to include row and column headers on printout of document
Record Body Length:
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Row and Column Header Print Flag	
	Do not print headers	0
	Print headers	1

PRINT GRIDLINES	2Bh	43d

Record Type: PRINT GRIDLINES
Description: Flag determines whether to print gridlines on print-out of document
Record Body Length: 2
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Gridline Print Flag	
	Do not print gridlines	0
	Print gridlines	1

FILEPASS	2Fh	47d

Record Type: FILEPASS
Description: Specifies a file password. If this record is present, the rest of the file is encrypted. The file password specified here is distinct from the document password specified by the PASSWORD record. If present, the FILEPASS record must immediately follow the BOF record.
Record Body Length: ?
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-on	?	

FONT	31h	49d

Record Type: FONT
Description: Describes an entry in the document's font table. A document may have up to 4 different fonts, numbered 0 to 3. Font records are written in the font table in the order in which they are encountered in the file.
Record Body Length: variable
Record Body Byte Structure:

Byte Number	Byte Description	Contents (binary)
0-1	Height of the font (in 1/20ths of a point)	
2-3	Font Attributes	
	First byte (reserved)	00000000b
	Second byte	
	Bit 0 - bold	1b
	Bit 1 - italic	1b
	Bit 2 - underline	1b
	Bit 2 - strikeout	1b
	Bits 4-7 (reserved)	0000b
4	Length of font name	
5-?	Font name	

FONT2 32h 50d

Record Type: FONT2
Description: System specific information about the font defined in the previous FONT record. The FONT2 record is option-al.
Record Body Length: Variable
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-on	?	

TABLE 36h 54d

Record Type: TABLE
Description: Describes a one-input row or column table created through the Data Table command
Record Body Length: 12 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	First row of the table	
2-3	Last row of the table	
4	First column of the table	
5	Last column of the table	
6	Recalculation flag	
	Table is recalculated	0
	Not recalculated	nonzero
7	Row or column input table flag	
	Column input table	0
	Row input table	1
8-9	Row of the input cell	
10-11	Column of the input cell	

The area given by the first and last rows and columns does not include the outer row or column, which contains table formulas or input values. If the input cell is a deleted reference, the row of the input cell, given by the bytes at offset 8 and 9, is -1.
TABLE2 37h 55d

Record Type: TABLE2
Description: Describes a two-input table created by the Data Table command. It is the same as the TABLE record, except there is no distinction between a row input table and a column input table, there are two input cells rather than one, and either may have a value of -1, indicating a deleted reference.
Record Body Length: 16 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	First row of the table	
2-3	Last row of the table	
4	First column of the table	
5	Last column of the table	
6	Recalculation flag	
	Table is calculated	0
	Needs recalculation	nonzero
7	RESERVED - must be zero	0
8-9	Row of the row input cell	
10-11	Column of the row input cell	
12-13	Row of the column input cell	
14-15	Column of the column input cell	

CONTINUE 3Ch 60d

Record Type: CONTINUE
Description: Continuation of FORMULA, ARRAY, or EXTERNNAME records that are too long to fit in a single record.
Record Body Length: variable
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-on	Parsed expression	

WINDOW1 3Dh 61d

Record Type: WINDOW1
Description: Basic window information. Locations are relative to the upper left corner of the Microsoft Windows desktop, and are measured in units of 1/20th of a point.
Record Body Length: 9 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Horizontal position of the window	
2-3	Vertical position of the window	
4-5	Width of the window	
6-7	Height of the window	
8	Hidden attribute	
	Window is not hidden	0
	Window is hidden	1

If you do not include a WINDOW1 record in your BIFF file, Excel will create a default window in your document.
WINDOW2 3Eh 62d

Record Type: WINDOW2
Description: Advanced window information. The WINDOW2 record is optional. If present, it must immediately follow the WINDOW1 record.
Record Body Length: 14 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0	Display Formulas	
	Display values	0
	Display formulas	1
1	Display Grid	
	Do not display gridlines	0
	Display gridlines	1
2	Display Row and Column Headers	
	Do not display headers	0
	Display headers	
3	Freeze window panes	
	Do not freeze panes	0
	Freeze panes	1
4	Display zero values	
	Suppress display	0
	Display zero values	1
5-6	Top row visible in the window	
7-8	Leftmost column visible in the window	
9	Row/column header and gridline color	
	Specified in next four bytes	0
	Use window's default foreground color.	1
10-13	Row/column headers and gridline color (RGB)	

BACKUP 40h 64d

Record Type: BACKUP
Description: Specifies whether a BIFF file should be backed up
Record Body Length: 2 bytes
Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Backup flag	
	Do not back up	0
	Back up	1

PANE 41h 65d

Record Type: PANE
Description: Describes the number and position of unfrozen panes in a window. Panes are created by horizontal and vertical splits, which are measured in units of 1/20th of a point.
Record Body Length: 9 bytes

Record Body Byte Structure:

Byte Number	Byte Description	Contents (hex)
0-1	Horizontal position of the split, zero if none	
2-3	Vertical position of the split, zero if none	
4-5	Top row visible in the bottom pane	
6-7	Leftmost column visible in the right pane	
8	Pane number of the active pane	

EDITORIAL NOTE:

This version was downloaded from the file Appnote.txt
from the PKZip Website on July 13, 1998.

To obtain this file go to:
http://www.pkware.com/download.html

Then download --> appnote.zip

END OF EDITORIAL NOTE

Disclaimer

Although PKWARE will attempt to supply current and accurate information relating to its file formats, algorithms, and the subject programs, the possibility of error can not be eliminated. PKWARE therefore expressly disclaims any warranty that the information contained in the associated materials relating to the subject programs and/or the format of the files created or accessed by the subject programs and/or the algorithms used by the subject programs, or any other matter, is current, correct or accurate as delivered. Any risk of damage due to any possible inaccurate information is assumed by the user of the information. Furthermore, the information relating to the subject programs and/or the file formats created or accessed by the subject programs and/or the algorithms used by the subject programs is subject to change without notice.

General Format of a ZIP file

Files stored in arbitrary order. Large zipfiles can span multiple diskette media.

Overall zipfile format:

```
  [local file header + file data + data_descriptor] . . .
  [central directory] end of central directory record
```

A. Local file header:

```
    local file header signature     4 bytes  (0x04034b50)
    version needed to extract       2 bytes
    general purpose bit flag        2 bytes
    compression method              2 bytes
    last mod file time              2 bytes
    last mod file date              2 bytes
    crc-32                          4 bytes
    compressed size                 4 bytes
    uncompressed size               4 bytes
    filename length                 2 bytes
    extra field length              2 bytes

    filename (variable size)
    extra field (variable size)
```

B. Data descriptor:

```
    crc-32                          4 bytes
    compressed size                 4 bytes
    uncompressed size               4 bytes
```

This descriptor exists only if bit 3 of the general purpose bit flag is set (see below). It is byte aligned and immediately follows the last byte of compressed data.

This descriptor is used only when it was not possible to seek in the output zip file, e.g., when the output zip file was standard output or a non seekable device.

C. Central directory structure:

```
[file header] . . .  end of central dir record
```

File header:

```
central file header signature   4 bytes  (0x02014b50)
version made by                 2 bytes
version needed to extract       2 bytes
general purpose bit flag        2 bytes
compression method              2 bytes
last mod file time              2 bytes
last mod file date              2 bytes
crc-32                          4 bytes
compressed size                 4 bytes
uncompressed size               4 bytes
filename length                 2 bytes
extra field length              2 bytes
file comment length             2 bytes
disk number start               2 bytes
internal file attributes        2 bytes
external file attributes        4 bytes
relative offset of local header 4 bytes

filename (variable size)
extra field (variable size)
file comment (variable size)
```

End of central dir record:

```
end of central dir signature    4 bytes  (0x06054b50)
number of this disk             2 bytes
number of the disk with the
start of the central directory  2 bytes
total number of entries in
the central dir on this disk    2 bytes
total number of entries in
the central dir                 2 bytes
size of the central directory   4 bytes
offset of start of central
directory with respect to
the starting disk number        4 bytes
zipfile comment length          2 bytes
zipfile comment (variable size)
```

D. Explanation of fields:

version made by (2 bytes)

The upper byte indicates the compatibility of the file attribute information. If the external file attributes are compatible with MS-DOS and can be read by PKZIP for DOS version 2.04g then this value will be zero. If these attributes are not compatible, then this value will identify the host system on which the attributes are compatible. Software can use this information to determine the line record format for text files etc. The current mappings are:

```
 0 - MS-DOS and OS/2 (FAT / VFAT / FAT32 file systems)
 1 - Amiga                     2 - VAX/VMS
 3 - Unix                      4 - VM/CMS
 5 - Atari ST                  6 - OS/2 H.P.F.S.
 7 - Macintosh                 8 - Z-System
 9 - CP/M                     10 - Windows NTFS
11 thru 255 - unused
```

The lower byte indicates the version number of the

```
    software used to encode the file.  The value/10
    indicates the major version number, and the value
    mod 10 is the minor version number.

version needed to extract (2 bytes)

    The minimum software version needed to extract the
    file, mapped as above.

general purpose bit flag: (2 bytes)

    Bit 0: If set, indicates that the file is encrypted.

    (For Method 6 - Imploding)
    Bit 1: If the compression method used was type 6,
           Imploding, then this bit, if set, indicates
           an 8K sliding dictionary was used.  If clear,
           then a 4K sliding dictionary was used.
    Bit 2: If the compression method used was type 6,
           Imploding, then this bit, if set, indicates
           an 3 Shannon-Fano trees were used to encode the
           sliding dictionary output.  If clear, then 2
           Shannon-Fano trees were used.

    (For Method 8 - Deflating)
    Bit 2  Bit 1
      0      0    Normal (-en) compression option was used.
      0      1    Maximum (-ex) compression option was used.
      1      0    Fast (-ef) compression option was used.
      1      1    Super Fast (-es) compression option was used.

    Note:  Bits 1 and 2 are undefined if the compression
           method is any other.

    Bit 3: If this bit is set, the fields crc-32, compressed size
           and uncompressed size are set to zero in the local
           header.  The correct values are put in the data descriptor
           immediately following the compressed data.  (Note: PKZIP
           version 2.04g for DOS only recognizes this bit for method 8
           compression, newer versions of PKZIP recognize this bit
           for any compression method.)

    The upper three bits are reserved and used internally
    by the software when processing the zipfile.  The
    remaining bits are unused.

compression method: (2 bytes)

    (see accompanying documentation for algorithm
    descriptions)

    0 - The file is stored (no compression)
    1 - The file is Shrunk
    2 - The file is Reduced with compression factor 1
    3 - The file is Reduced with compression factor 2
    4 - The file is Reduced with compression factor 3
    5 - The file is Reduced with compression factor 4
    6 - The file is Imploded
    7 - Reserved for Tokenizing compression algorithm
    8 - The file is Deflated
    9 - Reserved for enhanced Deflating
   10 - PKWARE Date Compression Library Imploding

date and time fields: (2 bytes each)

    The date and time are encoded in standard MS-DOS format.
    If input came from standard input, the date and time are
    those at which compression was started for this data.

CRC-32: (4 bytes)

    The CRC-32 algorithm was generously contributed by
    David Schwaderer and can be found in his excellent
```

```
    book "C Programmers Guide to NetBIOS" published by
    Howard W. Sams & Co. Inc.  The 'magic number' for
    the CRC is 0xdebb20e3.  The proper CRC pre and post
    conditioning is used, meaning that the CRC register
    is pre-conditioned with all ones (a starting value
    of 0xffffffff) and the value is post-conditioned by
    taking the one's complement of the CRC residual.
    If bit 3 of the general purpose flag is set, this
    field is set to zero in the local header and the correct
    value is put in the data descriptor and in the central
    directory.

compressed size: (4 bytes)
uncompressed size: (4 bytes)

    The size of the file compressed and uncompressed,
    respectively.  If bit 3 of the general purpose bit flag
    is set, these fields are set to zero in the local header
    and the correct values are put in the data descriptor and
    in the central directory.

filename length: (2 bytes)
extra field length: (2 bytes)
file comment length: (2 bytes)

    The length of the filename, extra field, and comment
    fields respectively.  The combined length of any
    directory record and these three fields should not
    generally exceed 65,535 bytes.  If input came from standard
    input, the filename length is set to zero.


disk number start: (2 bytes)

    The number of the disk on which this file begins.

internal file attributes: (2 bytes)

    The lowest bit of this field indicates, if set, that
    the file is apparently an ASCII or text file.  If not
    set, that the file apparently contains binary data.
    The remaining bits are unused in version 1.0.

external file attributes: (4 bytes)

    The mapping of the external attributes is
    host-system dependent (see 'version made by').  For
    MS-DOS, the low order byte is the MS-DOS directory
    attribute byte.  If input came from standard input, this
    field is set to zero.

relative offset of local header: (4 bytes)

    This is the offset from the start of the first disk on
    which this file appears, to where the local header should
    be found.

filename: (Variable)

    The name of the file, with optional relative path.
    The path stored should not contain a drive or
    device letter, or a leading slash.  All slashes
    should be forward slashes '/' as opposed to
    backwards slashes '\' for compatibility with Amiga
    and Unix file systems etc.  If input came from standard
    input, there is no filename field.

extra field: (Variable)

    This is for future expansion.  If additional information
    needs to be stored in the future, it should be stored
    here.  Earlier versions of the software can then safely
    skip this file, and find the next file or header.  This
```

field will be 0 length in version 1.0.

In order to allow different programs and different types of information to be stored in the 'extra' field in .ZIP files, the following structure should be used for all programs storing data in this field:

header1+data1 + header2+data2 . . .

Each header should consist of:

```
  Header ID - 2 bytes
  Data Size - 2 bytes
```

Note: all fields stored in Intel low-byte/high-byte order.

The Header ID field indicates the type of data that is in the following data block.

Header ID's of 0 thru 31 are reserved for use by PKWARE. The remaining ID's can be used by third party vendors for proprietary usage.

The current Header ID mappings defined by PKWARE are:

```
0x0007        AV Info
0x0009        OS/2
0x000c        VAX/VMS
0x000d        reserved for Unix
```

Several third party mappings commonly used are:

```
0x4b46        FWKCS MD5 (see below)
0x07c8        Macintosh
0x4341        Acorn/SparkFS
0x4453        Windows NT security descriptor (binary ACL)
0x4704        VM/CMS
0x470f        MVS
0x4c41        OS/2 access control list (text ACL)
0x4d49        Info-ZIP VMS (VAX or Alpha)
0x5455        extended timestamp
0x5855        Info-ZIP Unix (original, also OS/2, NT, etc)
0x6542        BeOS/BeBox
0x756e        ASi Unix
0x7855        Info-ZIP Unix (new)
0xfd4a        SMS/QDOS
```

The Data Size field indicates the size of the following data block. Programs can use this value to skip to the next header block, passing over any data blocks that are not of interest.

```
Note: As stated above, the size of the entire .ZIP file
      header, including the filename, comment, and extra
      field should not exceed 64K in size.
```

In case two different programs should appropriate the same Header ID value, it is strongly recommended that each program place a unique signature of at least two bytes in size (and preferably 4 bytes or bigger) at the start of each data area. Every program should verify that its unique signature is present, in addition to the Header ID value being correct, before assuming that it is a block of known type.

-OS/2 Extra Field:

The following is the layout of the OS/2 attributes "extra" block. (Last Revision 09/05/95)

Note: all fields stored in Intel low-byte/high-byte order.

```
        Value         Size            Description
        -----         ----            -----------
(OS/2)  0x0009        Short           Tag for this "extra" block type
        TSize         Short           Size for the following data block
        BSize         Long            Uncompressed Block Size
        CType         Short           Compression type
        EACRC         Long            CRC value for uncompress block
        (var)         variable        Compressed block
```

The OS/2 extended attribute structure (FEA2LIST) is compressed and then stored in it's entirety within this structure. There will only ever be one "block" of data in VarFields[].

-VAX/VMS Extra Field:

The following is the layout of the VAX/VMS attributes "extra" block. (Last Revision 12/17/91)

Note: all fields stored in Intel low-byte/high-byte order.

```
        Value         Size            Description
        -----         ----            -----------
(VMS)   0x000c        Short           Tag for this "extra" block type
        TSize         Short           Size of the total "extra" block
        CRC           Long            32-bit CRC for remainder of the block
        Tag1          Short           VMS attribute tag value #1
        Size1         Short           Size of attribute #1, in bytes
        (var.)        Size1           Attribute #1 data
        .
        .
        .
        TagN          Short           VMS attribute tage value #N
        SizeN         Short           Size of attribute #N, in bytes
        (var.)        SizeN           Attribute #N data
```

Rules:

1. There will be one or more of attributes present, which will each be preceded by the above TagX & SizeX values. These values are identical to the ATR$C_XXXX and ATR$S_XXXX constants which are defined in ATR.H under VMS C. Neither of these values will ever be zero.

2. No word alignment or padding is performed.

3. A well-behaved PKZIP/VMS program should never produce more than one sub-block with the same TagX value. Also, there will never be more than one "extra" block of type 0x000c in a particular directory record.

- FWKCS MD5 Extra Field:

The FWKCS Contents_Signature System, used in automatically identifying files independent of filename, optionally adds and uses an extra field to support the rapid creation of an enhanced contents_signature:

```
    Header ID = 0x4b46
    Data Size = 0x0013
    Preface   = 'M','D','5'
    followed by 16 bytes containing the uncompressed
        file's 128_bit MD5 hash(1), low byte first.
```

When FWKCS revises a zipfile central directory to add this extra field for a file, it also replaces the central directory entry for that file's uncompressed filelength with a measured value.

FWKCS provides an option to strip this extra field, if present, from a zipfile central directory. In adding this extra field, FWKCS preserves Zipfile Authenticity Verification; if stripping this extra field, FWKCS

preserves all versions of AV through PKZIP version 2.04g.

FWKCS, and FWKCS Contents_Signature System, are trademarks of Frederick W. Kantor.

(1) R. Rivest, RFC1321.TXT, MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992. ll.76-77: "The MD5 algorithm is being placed in the public domain for review and possible adoption as a standard."

file comment: (Variable)

The comment for this file.

number of this disk: (2 bytes)

The number of this disk, which contains central directory end record.

number of the disk with the start of the central directory: (2 bytes)

The number of the disk on which the central directory starts.

total number of entries in the central dir on this disk: (2 bytes)

The number of central directory entries on this disk.

total number of entries in the central dir: (2 bytes)

The total number of files in the zipfile.

size of the central directory: (4 bytes)

The size (in bytes) of the entire central directory.

offset of start of central directory with respect to the starting disk number: (4 bytes)

Offset of the start of the central directory on the disk on which the central directory starts.

zipfile comment length: (2 bytes)

The length of the comment for this zipfile.

zipfile comment: (Variable)

The comment for this zipfile.

D. General notes:

1) All fields unless otherwise noted are unsigned and stored in Intel low-byte:high-byte, low-word:high-word order.

2) String fields are not null terminated, since the length is given explicitly.

3) Local headers should not span disk boundaries. Also, even though the central directory can span disk boundaries, no single record in the central directory should be split across disks.

4) The entries in the central directory may not necessarily be in the same order that files appear in the zipfile.

UnShrinking - Method 1

Shrinking is a Dynamic Ziv-Lempel-Welch compression algorithm

with partial clearing. The initial code size is 9 bits, and the maximum code size is 13 bits. Shrinking differs from conventional Dynamic Ziv-Lempel-Welch implementations in several respects:

1) The code size is controlled by the compressor, and is not automatically increased when codes larger than the current code size are created (but not necessarily used). When the decompressor encounters the code sequence 256 (decimal) followed by 1, it should increase the code size read from the input stream to the next bit size. No blocking of the codes is performed, so the next code at the increased size should be read from the input stream immediately after where the previous code at the smaller bit size was read. Again, the decompressor should not increase the code size used until the sequence 256,1 is encountered.

2) When the table becomes full, total clearing is not performed. Rather, when the compressor emits the code sequence 256,2 (decimal), the decompressor should clear all leaf nodes from the Ziv-Lempel tree, and continue to use the current code size. The nodes that are cleared from the Ziv-Lempel tree are then re-used, with the lowest code value re-used first, and the highest code value re-used last. The compressor can emit the sequence 256,2 at any time.

Expanding - Methods 2-5

The Reducing algorithm is actually a combination of two distinct algorithms. The first algorithm compresses repeated byte sequences, and the second algorithm takes the compressed stream from the first algorithm and applies a probabilistic compression method.

The probabilistic compression stores an array of 'follower sets' S(j), for j=0 to 255, corresponding to each possible ASCII character. Each set contains between 0 and 32 characters, to be denoted as S(j)[0],...,S(j)[m], where m<32. The sets are stored at the beginning of the data area for a Reduced file, in reverse order, with S(255) first, and S(0) last.

The sets are encoded as { N(j), S(j)[0],...,S(j)[N(j)-1] }, where N(j) is the size of set S(j). N(j) can be 0, in which case the follower set for S(j) is empty. Each N(j) value is encoded in 6 bits, followed by N(j) eight bit character values corresponding to S(j)[0] to S(j)[N(j)-1] respectively. If N(j) is 0, then no values for S(j) are stored, and the value for N(j-1) immediately follows.

Immediately after the follower sets, is the compressed data stream. The compressed data stream can be interpreted for the probabilistic decompression as follows:

```
let Last-Character <- 0.
loop until done
    if the follower set S(Last-Character) is empty then
        read 8 bits from the input stream, and copy this
        value to the output stream.
    otherwise if the follower set S(Last-Character) is non-empty then
        read 1 bit from the input stream.
        if this bit is not zero then
            read 8 bits from the input stream, and copy this
            value to the output stream.
        otherwise if this bit is zero then
            read B(N(Last-Character)) bits from the input
            stream, and assign this value to I.
```

```
            Copy the value of S(Last-Character)[I] to the
            output stream.

    assign the last value placed on the output stream to
    Last-Character.
end loop
```

B(N(j)) is defined as the minimal number of bits required to
encode the value N(j)-1.

The decompressed stream from above can then be expanded to
re-create the original file as follows:

```
let State <- 0.

loop until done
    read 8 bits from the input stream into C.
    case State of
        0:  if C is not equal to DLE (144 decimal) then
                copy C to the output stream.
            otherwise if C is equal to DLE then
                let State <- 1.

        1:  if C is non-zero then
                let V <- C.
                let Len <- L(V)
                let State <- F(Len).
            otherwise if C is zero then
                copy the value 144 (decimal) to the output stream.
                let State <- 0

        2:  let Len <- Len + C
            let State <- 3.

        3:  move backwards D(V,C) bytes in the output stream
            (if this position is before the start of the output
            stream, then assume that all the data before the
            start of the output stream is filled with zeros).
            copy Len+3 bytes from this position to the output stream.
            let State <- 0.
    end case
end loop
```

The functions F,L, and D are dependent on the 'compression
factor', 1 through 4, and are defined as follows:

```
For compression factor 1:
    L(X) equals the lower 7 bits of X.
    F(X) equals 2 if X equals 127 otherwise F(X) equals 3.
    D(X,Y) equals the (upper 1 bit of X) * 256 + Y + 1.
For compression factor 2:
    L(X) equals the lower 6 bits of X.
    F(X) equals 2 if X equals 63 otherwise F(X) equals 3.
    D(X,Y) equals the (upper 2 bits of X) * 256 + Y + 1.
For compression factor 3:
    L(X) equals the lower 5 bits of X.
    F(X) equals 2 if X equals 31 otherwise F(X) equals 3.
    D(X,Y) equals the (upper 3 bits of X) * 256 + Y + 1.
For compression factor 4:
    L(X) equals the lower 4 bits of X.
    F(X) equals 2 if X equals 15 otherwise F(X) equals 3.
    D(X,Y) equals the (upper 4 bits of X) * 256 + Y + 1.
```

Imploding - Method 6

The Imploding algorithm is actually a combination of two distinct
algorithms. The first algorithm compresses repeated byte

sequences using a sliding dictionary. The second algorithm is used to compress the encoding of the sliding dictionary output, using multiple Shannon-Fano trees.

The Imploding algorithm can use a 4K or 8K sliding dictionary size. The dictionary size used can be determined by bit 1 in the general purpose flag word; a 0 bit indicates a 4K dictionary while a 1 bit indicates an 8K dictionary.

The Shannon-Fano trees are stored at the start of the compressed file. The number of trees stored is defined by bit 2 in the general purpose flag word; a 0 bit indicates two trees stored, a 1 bit indicates three trees are stored. If 3 trees are stored, the first Shannon-Fano tree represents the encoding of the Literal characters, the second tree represents the encoding of the Length information, the third represents the encoding of the Distance information. When 2 Shannon-Fano trees are stored, the Length tree is stored first, followed by the Distance tree.

The Literal Shannon-Fano tree, if present is used to represent the entire ASCII character set, and contains 256 values. This tree is used to compress any data not compressed by the sliding dictionary algorithm. When this tree is present, the Minimum Match Length for the sliding dictionary is 3. If this tree is not present, the Minimum Match Length is 2.

The Length Shannon-Fano tree is used to compress the Length part of the (length,distance) pairs from the sliding dictionary output. The Length tree contains 64 values, ranging from the Minimum Match Length, to 63 plus the Minimum Match Length.

The Distance Shannon-Fano tree is used to compress the Distance part of the (length,distance) pairs from the sliding dictionary output. The Distance tree contains 64 values, ranging from 0 to 63, representing the upper 6 bits of the distance value. The distance values themselves will be between 0 and the sliding dictionary size, either 4K or 8K.

The Shannon-Fano trees themselves are stored in a compressed format. The first byte of the tree data represents the number of bytes of data representing the (compressed) Shannon-Fano tree minus 1. The remaining bytes represent the Shannon-Fano tree data encoded as:

```
High 4 bits: Number of values at this bit length + 1. (1 - 16)
Low  4 bits: Bit Length needed to represent value + 1. (1 - 16)
```

The Shannon-Fano codes can be constructed from the bit lengths using the following algorithm:

1) Sort the Bit Lengths in ascending order, while retaining the order of the original lengths stored in the file.

2) Generate the Shannon-Fano trees:

```
Code <- 0
CodeIncrement <- 0
LastBitLength <- 0
i <- number of Shannon-Fano codes - 1   (either 255 or 63)

loop while i >= 0
    Code = Code + CodeIncrement
    if BitLength(i) <> LastBitLength then
        LastBitLength=BitLength(i)
        CodeIncrement = 1 shifted left (16 - LastBitLength)
    ShannonCode(i) = Code
    i <- i - 1
end loop
```

3) Reverse the order of all the bits in the above ShannonCode() vector, so that the most significant bit becomes the least significant bit. For example, the value 0x1234 (hex) would

become 0x2C48 (hex).

4) Restore the order of Shannon-Fano codes as originally stored within the file.

Example:

This example will show the encoding of a Shannon-Fano tree of size 8. Notice that the actual Shannon-Fano trees used for Imploding are either 64 or 256 entries in size.

Example: 0x02, 0x42, 0x01, 0x13

The first byte indicates 3 values in this table. Decoding the bytes:

```
        0x42 = 5 codes of 3 bits long
        0x01 = 1 code  of 2 bits long
        0x13 = 2 codes of 4 bits long
```

This would generate the original bit length array of:
(3, 3, 3, 3, 3, 2, 4, 4)

There are 8 codes in this table for the values 0 thru 7. Using the algorithm to obtain the Shannon-Fano codes produces:

Val	Sorted	Constructed Code	Reversed Value	Order Restored	Original Length
0:	2	1100000000000000	11	101	3
1:	3	1010000000000000	101	001	3
2:	3	1000000000000000	001	110	3
3:	3	0110000000000000	110	010	3
4:	3	0100000000000000	010	100	3
5:	3	0010000000000000	100	11	2
6:	4	0001000000000000	1000	1000	4
7:	4	0000000000000000	0000	0000	4

The values in the Val, Order Restored and Original Length columns now represent the Shannon-Fano encoding tree that can be used for decoding the Shannon-Fano encoded data. How to parse the variable length Shannon-Fano values from the data stream is beyond the scope of this document. (See the references listed at the end of this document for more information.) However, traditional decoding schemes used for Huffman variable length decoding, such as the Greenlaw algorithm, can be successfully applied.

The compressed data stream begins immediately after the compressed Shannon-Fano data. The compressed data stream can be interpreted as follows:

```
loop until done
    read 1 bit from input stream.

    if this bit is non-zero then       (encoded data is literal data)
        if Literal Shannon-Fano tree is present
            read and decode character using Literal Shannon-Fano tree.
        otherwise
            read 8 bits from input stream.
        copy character to the output stream.
    otherwise                   (encoded data is sliding dictionary match)
        if 8K dictionary size
            read 7 bits for offset Distance (lower 7 bits of offset).
        otherwise
            read 6 bits for offset Distance (lower 6 bits of offset).

        using the Distance Shannon-Fano tree, read and decode the
          upper 6 bits of the Distance value.

        using the Length Shannon-Fano tree, read and decode
          the Length value.

        Length <- Length + Minimum Match Length
```

```
        if Length = 63 + Minimum Match Length
            read 8 bits from the input stream,
            add this value to Length.

        move backwards Distance+1 bytes in the output stream, and
        copy Length characters from this position to the output
        stream.  (if this position is before the start of the output
        stream, then assume that all the data before the start of
        the output stream is filled with zeros).
end loop
```

Tokenizing - Method 7

This method is not used by PKZIP.

Deflating - Method 8

The Deflate algorithm is similar to the Implode algorithm using
a sliding dictionary of up to 32K with secondary compression
from Huffman/Shannon-Fano codes.

The compressed data is stored in blocks with a header describing
the block and the Huffman codes used in the data block. The header
format is as follows:

```
   Bit 0: Last Block bit     This bit is set to 1 if this is the last
                             compressed block in the data.
   Bits 1-2: Block type
      00 (0) - Block is stored - All stored data is byte aligned.
               Skip bits until next byte, then next word = block length,
               followed by the ones compliment of the block length word.
               Remaining data in block is the stored data.

      01 (1) - Use fixed Huffman codes for literal and distance codes.
               Lit Code    Bits             Dist Code   Bits
               ---------   ----             ---------   ----
                 0 - 143    8                 0 - 31      5
               144 - 255    9
               256 - 279    7
               280 - 287    8

               Literal codes 286-287 and distance codes 30-31 are never
               used but participate in the huffman construction.

      10 (2) - Dynamic Huffman codes.  (See expanding Huffman codes)

      11 (3) - Reserved - Flag a "Error in compressed data" if seen.
```

Expanding Huffman Codes

If the data block is stored with dynamic Huffman codes, the Huffman
codes are sent in the following compressed format:

```
   5 Bits: # of Literal codes sent - 256 (256 - 286)
           All other codes are never sent.
   5 Bits: # of Dist codes - 1           (1 - 32)
   4 Bits: # of Bit Length codes - 3     (3 - 19)
```

The Huffman codes are sent as bit lengths and the codes are built as
described in the implode algorithm. The bit lengths themselves are
compressed with Huffman codes. There are 19 bit length codes:

```
   0 - 15: Represent bit lengths of 0 - 15
       16: Copy the previous bit length 3 - 6 times.
           The next 2 bits indicate repeat length (0 = 3, ... ,3 = 6)
              Example:  Codes 8, 16 (+2 bits 11), 16 (+2 bits 10) will
                        expand to 12 bit lengths of 8 (1 + 6 + 5)
       17: Repeat a bit length of 0 for 3 - 10 times. (3 bits of length)
       18: Repeat a bit length of 0 for 11 - 138 times (7 bits of length)
```

The lengths of the bit length codes are sent packed 3 bits per value (0 - 7) in the following order:

16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15

The Huffman codes should be built as described in the Implode algorithm except codes are assigned starting at the shortest bit length, i.e. the shortest code should be all 0's rather than all 1's. Also, codes with a bit length of zero do not participate in the tree construction. The codes are then used to decode the bit lengths for the literal and distance tables.

The bit lengths for the literal tables are sent first with the number of entries sent described by the 5 bits sent earlier. There are up to 286 literal characters; the first 256 represent the respective 8 bit character, code 256 represents the End-Of-Block code, the remaining 29 codes represent copy lengths of 3 thru 258. There are up to 30 distance codes representing distances from 1 thru 32k as described below.

Length Codes

Code	Extra Bits	Length	Code	Extra Bits	Lengths	Code	Extra Bits	Lengths	Code	Extra Bits	Length(s)
257	0	3	265	1	11,12	273	3	35-42	281	5	131-162
258	0	4	266	1	13,14	274	3	43-50	282	5	163-194
259	0	5	267	1	15,16	275	3	51-58	283	5	195-226
260	0	6	268	1	17,18	276	3	59-66	284	5	227-257
261	0	7	269	2	19-22	277	4	67-82	285	0	258
262	0	8	270	2	23-26	278	4	83-98			
263	0	9	271	2	27-30	279	4	99-114			
264	0	10	272	2	31-34	280	4	115-130			

Distance Codes

Code	Extra Bits	Dist	Code	Extra Bits	Dist	Code	Extra Bits	Distance	Code	Extra Bits	Distance
0	0	1	8	3	17-24	16	7	257-384	24	11	4097-6144
1	0	2	9	3	25-32	17	7	385-512	25	11	6145-8192
2	0	3	10	4	33-48	18	8	513-768	26	12	8193-12288
3	0	4	11	4	49-64	19	8	769-1024	27	12	12289-16384
4	1	5,6	12	5	65-96	20	9	1025-1536	28	13	16385-24576
5	1	7,8	13	5	97-128	21	9	1537-2048	29	13	24577-32768
6	2	9-12	14	6	129-192	22	10	2049-3072			
7	2	13-16	15	6	193-256	23	10	3073-4096			

The compressed data stream begins immediately after the compressed header data. The compressed data stream can be interpreted as follows:

```
do
   read header from input stream.

   if stored block
      skip bits until byte aligned
      read count and 1's compliment of count
      copy count bytes data block
   otherwise
      loop until end of block code sent
         decode literal character from input stream
         if literal < 256
            copy character to the output stream
         otherwise
            if literal = end of block
               break from loop
            otherwise
               decode distance from input stream

               move backwards distance bytes in the output stream, and
               copy length characters from this position to the output
               stream.
```

```
      end loop
while not last block

if data descriptor exists
   skip bits until byte aligned
   read crc and sizes
endif
```

Decryption

The encryption used in PKZIP was generously supplied by Roger Schlafly. PKWARE is grateful to Mr. Schlafly for his expert help and advice in the field of data encryption.

PKZIP encrypts the compressed data stream. Encrypted files must be decrypted before they can be extracted.

Each encrypted file has an extra 12 bytes stored at the start of the data area defining the encryption header for that file. The encryption header is originally set to random values, and then itself encrypted, using three, 32-bit keys. The key values are initialized using the supplied encryption password. After each byte is encrypted, the keys are then updated using pseudo-random number generation techniques in combination with the same CRC-32 algorithm used in PKZIP and described elsewhere in this document.

The following is the basic steps required to decrypt a file:

1) Initialize the three 32-bit keys with the password.
2) Read and decrypt the 12-byte encryption header, further initializing the encryption keys.
3) Read and decrypt the compressed data stream using the encryption keys.

Step 1 - Initializing the encryption keys

```
Key(0) <- 305419896
Key(1) <- 591751049
Key(2) <- 878082192

loop for i <- 0 to length(password)-1
    update_keys(password(i))
end loop
```

Where update_keys() is defined as:

```
update_keys(char):
  Key(0) <- crc32(key(0),char)
  Key(1) <- Key(1) + (Key(0) & 000000ffH)
  Key(1) <- Key(1) * 134775813 + 1
  Key(2) <- crc32(key(2),key(1) >> 24)
end update_keys
```

Where crc32(old_crc,char) is a routine that given a CRC value and a character, returns an updated CRC value after applying the CRC-32 algorithm described elsewhere in this document.

Step 2 - Decrypting the encryption header

The purpose of this step is to further initialize the encryption keys, based on random data, to render a plaintext attack on the data ineffective.

Read the 12-byte encryption header into Buffer, in locations

Buffer(0) thru Buffer(11).

```
loop for i <- 0 to 11
    C <- buffer(i) ^ decrypt_byte()
    update_keys(C)
    buffer(i) <- C
end loop
```

Where decrypt_byte() is defined as:

```
unsigned char decrypt_byte()
    local unsigned short temp
    temp <- Key(2) | 2
    decrypt_byte <- (temp * (temp ^ 1)) >> 8
end decrypt_byte
```

After the header is decrypted, the last 1 or 2 bytes in Buffer should be the high-order word/byte of the CRC for the file being decrypted, stored in Intel low-byte/high-byte order. Versions of PKZIP prior to 2.0 used a 2 byte CRC check; a 1 byte CRC check is used on versions after 2.0. This can be used to test if the password supplied is correct or not.

Step 3 - Decrypting the compressed data stream

The compressed data stream can be decrypted as follows:

```
loop until done
    read a character into C
    Temp <- C ^ decrypt_byte()
    update_keys(temp)
    output Temp
end loop
```

In addition to the above mentioned contributors to PKZIP and PKUNZIP, I would like to extend special thanks to Robert Mahoney for suggesting the extension .ZIP for this software.

References:

Fiala, Edward R., and Greene, Daniel H., "Data compression with finite windows", Communications of the ACM, Volume 32, Number 4, April 1989, pages 490-505.

Held, Gilbert, "Data Compression, Techniques and Applications, Hardware and Software Considerations", John Wiley & Sons, 1987.

Huffman, D.A., "A method for the construction of minimum-redundancy codes", Proceedings of the IRE, Volume 40, Number 9, September 1952, pages 1098-1101.

Nelson, Mark, "LZW Data Compression", Dr. Dobbs Journal, Volume 14, Number 10, October 1989, pages 29-37.

Nelson, Mark, "The Data Compression Book", M&T Books, 1991.

Storer, James A., "Data Compression, Methods and Theory", Computer Science Press, 1988

Welch, Terry, "A Technique for High-Performance Data Compression", IEEE Computer, Volume 17, Number 6, June 1984, pages 8-19.

Ziv, J. and Lempel, A., "A universal algorithm for sequential data compression", Communications of the ACM, Volume 30, Number 6,

June 1987, pages 520-540.

Ziv, J. and Lempel, A., "Compression of individual sequences via variable-rate coding", IEEE Transactions on Information Theory, Volume 24, Number 5, September 1978, pages 530-536.

File: APPNOTE.TXT - .ZIP File Format Specification
Version: 4.5
Revised: 11/01/2001
Copyright (c) 1989 - 2001 PKWARE Inc., All Rights Reserved.

Disclaimer

Although PKWARE will attempt to supply current and accurate information relating to its file formats, algorithms, and the subject programs, the possibility of error can not be eliminated. PKWARE therefore expressly disclaims any warranty that the information contained in the associated materials relating to the subject programs and/or the format of the files created or accessed by the subject programs and/or the algorithms used by the subject programs, or any other matter, is current, correct or accurate as delivered. Any risk of damage due to any possible inaccurate information is assumed by the user of the information. Furthermore, the information relating to the subject programs and/or the file formats created or accessed by the subject programs and/or the algorithms used by the subject programs is subject to change without notice.

General Format of a .ZIP file

Files stored in arbitrary order. Large .ZIP files can span multiple diskette media or be split into user-defined segment sizes.

Overall .ZIP file format:

```
[local file header 1]
[file data 1]
[data descriptor 1]
.
.
.
[local file header n]
[file data n]
[data descriptor n]
[central directory]
[zip64 end of central directory record]
[zip64 end of central directory locator]
[end of central directory record]
```

A. Local file header:

```
local file header signature     4 bytes  (0x04034b50)
version needed to extract       2 bytes
general purpose bit flag        2 bytes
compression method              2 bytes
last mod file time              2 bytes
last mod file date              2 bytes
crc-32                          4 bytes
compressed size                 4 bytes
uncompressed size               4 bytes
file name length                2 bytes
extra field length              2 bytes

file name (variable size)
extra field (variable size)
```

B. File data

Immediately following the local header for a file is the compressed or stored data for the file. The series of [local file header][file data][data descriptor] repeats for each file in the .ZIP archive.

C. Data descriptor:

```
crc-32                          4 bytes
```

```
      compressed size                 4 bytes
      uncompressed size               4 bytes

    This descriptor exists only if bit 3 of the general
    purpose bit flag is set (see below).  It is byte aligned
    and immediately follows the last byte of compressed data.
    This descriptor is used only when it was not possible to
    seek in the output .ZIP file, e.g., when the output .ZIP file
    was standard output or a non seekable device.  For Zip64 format
    archives, the compressed and uncompressed sizes are 8 bytes each.

D.  Central directory structure:

    [file header 1]
    .
    .
    .
    [file header n]
    [digital signature]

    File header:

      central file header signature   4 bytes  (0x02014b50)
      version made by                 2 bytes
      version needed to extract       2 bytes
      general purpose bit flag        2 bytes
      compression method              2 bytes
      last mod file time              2 bytes
      last mod file date              2 bytes
      crc-32                          4 bytes
      compressed size                 4 bytes
      uncompressed size               4 bytes
      file name length                2 bytes
      extra field length              2 bytes
      file comment length             2 bytes
      disk number start               2 bytes
      internal file attributes        2 bytes
      external file attributes        4 bytes
      relative offset of local header 4 bytes

      file name (variable size)
      extra field (variable size)
      file comment (variable size)

    Digital signature:

      header signature                4 bytes  (0x05054b50)
      size of data                    2 bytes
      signature data (variable size)

E.  Zip64 end of central directory record

      zip64 end of central dir
      signature                       4 bytes  (0x06064b50)
      size of zip64 end of central
      directory record                8 bytes
      version made by                 2 bytes
      version needed to extract       2 bytes
      number of this disk             4 bytes
      number of the disk with the
      start of the central directory  4 bytes
      total number of entries in the
      central directory on this disk  8 bytes
      total number of entries in the
      central directory               8 bytes
      size of the central directory   8 bytes
      offset of start of central
      directory with respect to
      the starting disk number        8 bytes
      zip64 extensible data sector    (variable size)

F.  Zip64 end of central directory locator
```

```
      zip64 end of central dir locator
      signature                       4 bytes  (0x07064b50)
      number of the disk with the
      start of the zip64 end of
      central directory               4 bytes
      relative offset of the zip64
      end of central directory record 8 bytes
      total number of disks           4 bytes

G.  End of central directory record:

      end of central dir signature    4 bytes  (0x06054b50)
      number of this disk             2 bytes
      number of the disk with the
      start of the central directory  2 bytes
      total number of entries in the
      central directory on this disk  2 bytes
      total number of entries in
      the central directory           2 bytes
      size of the central directory   4 bytes
      offset of start of central
      directory with respect to
      the starting disk number        4 bytes
      .ZIP file comment length        2 bytes
      .ZIP file comment       (variable size)

H.  Explanation of fields:

    version made by (2 bytes)

        The upper byte indicates the compatibility of the file
        attribute information.  If the external file attributes
        are compatible with MS-DOS and can be read by PKZIP for
        DOS version 2.04g then this value will be zero.  If these
        attributes are not compatible, then this value will
        identify the host system on which the attributes are
        compatible.  Software can use this information to determine
        the line record format for text files etc.  The current
        mappings are:

        0 - MS-DOS and OS/2 (FAT / VFAT / FAT32 file systems)
        1 - Amiga                     2 - OpenVMS
        3 - Unix                      4 - VM/CMS
        5 - Atari ST                  6 - OS/2 H.P.F.S.
        7 - Macintosh                 8 - Z-System
        9 - CP/M                     10 - Windows NTFS
       11 - MVS                      12 - VSE
       13 - Acorn Risc               14 - VFAT
       15 - alternate MVS            16 - BeOS
       17 - Tandem
       18 thru 255 - unused

        The lower byte indicates the version number of the
        software used to encode the file.  The value/10
        indicates the major version number, and the value
        mod 10 is the minor version number.

    version needed to extract (2 bytes)

        The minimum software version needed to extract the
        file, mapped as above.  For Zip64 format archives,
        this value should not be less than 45.

    general purpose bit flag: (2 bytes)

        Bit 0: If set, indicates that the file is encrypted.

        (For Method 6 - Imploding)
        Bit 1: If the compression method used was type 6,
               Imploding, then this bit, if set, indicates
               an 8K sliding dictionary was used.  If clear,
               then a 4K sliding dictionary was used.
        Bit 2: If the compression method used was type 6,
```

```
        Imploding, then this bit, if set, indicates
        3 Shannon-Fano trees were used to encode the
        sliding dictionary output.  If clear, then 2
        Shannon-Fano trees were used.

    (For Methods 8 and 9 - Deflating)
    Bit 2  Bit 1
      0      0    Normal (-en) compression option was used.
      0      1    Maximum (-exx/-ex) compression option was used.
      1      0    Fast (-ef) compression option was used.
      1      1    Super Fast (-es) compression option was used.

    Note:  Bits 1 and 2 are undefined if the compression
           method is any other.

    Bit 3: If this bit is set, the fields crc-32, compressed
           size and uncompressed size are set to zero in the
           local header.  The correct values are put in the
           data descriptor immediately following the compressed
           data.  (Note: PKZIP version 2.04g for DOS only
           recognizes this bit for method 8 compression, newer
           versions of PKZIP recognize this bit for any
           compression method.)

    Bit 4: Reserved for use with method 8, for enhanced
           deflating.

    Bit 5: If this bit is set, this indicates that the file is
           compressed patched data.  (Note: Requires PKZIP
           version 2.70 or greater)

    Bit 6: Currently unused.

    Bit 7: Currently unused.

    Bit 8: Currently unused.

    Bit 9: Currently unused.

    Bit 10: Currently unused.

    Bit 11: Currently unused.

    Bit 12: Reserved by PKWARE for enhanced compression.

    Bit 13: Reserved by PKWARE.

    Bit 14: Reserved by PKWARE.

    Bit 15: Reserved by PKWARE.

compression method: (2 bytes)

    (see accompanying documentation for algorithm
    descriptions)

    0 - The file is stored (no compression)
    1 - The file is Shrunk
    2 - The file is Reduced with compression factor 1
    3 - The file is Reduced with compression factor 2
    4 - The file is Reduced with compression factor 3
    5 - The file is Reduced with compression factor 4
    6 - The file is Imploded
    7 - Reserved for Tokenizing compression algorithm
    8 - The file is Deflated
    9 - Enhanced Deflating using Deflate64(tm)
   10 - PKWARE Date Compression Library Imploding

date and time fields: (2 bytes each)

    The date and time are encoded in standard MS-DOS format.
    If input came from standard input, the date and time are
    those at which compression was started for this data.
```

CRC-32: (4 bytes)

The CRC-32 algorithm was generously contributed by David Schwaderer and can be found in his excellent book "C Programmers Guide to NetBIOS" published by Howard W. Sams & Co. Inc. The 'magic number' for the CRC is 0xdebb20e3. The proper CRC pre and post conditioning is used, meaning that the CRC register is pre-conditioned with all ones (a starting value of 0xffffffff) and the value is post-conditioned by taking the one's complement of the CRC residual. If bit 3 of the general purpose flag is set, this field is set to zero in the local header and the correct value is put in the data descriptor and in the central directory.

compressed size: (4 bytes)
uncompressed size: (4 bytes)

The size of the file compressed and uncompressed, respectively. If bit 3 of the general purpose bit flag is set, these fields are set to zero in the local header and the correct values are put in the data descriptor and in the central directory. If an archive is in zip64 format and the value in this field is 0xFFFFFFFF, the size will be in the corresponding 8 byte zip64 extended information extra field.

file name length: (2 bytes)
extra field length: (2 bytes)
file comment length: (2 bytes)

The length of the file name, extra field, and comment fields respectively. The combined length of any directory record and these three fields should not generally exceed 65,535 bytes. If input came from standard input, the file name length is set to zero.

disk number start: (2 bytes)

The number of the disk on which this file begins. If an archive is in zip64 format and the value in this field is 0xFFFF, the size will be in the corresponding 4 byte zip64 extended information extra field.

internal file attributes: (2 bytes)

The lowest bit of this field indicates, if set, that the file is apparently an ASCII or text file. If not set, that the file apparently contains binary data. The remaining bits are unused in version 1.0.

Bits 1 and 2 are reserved for use by PKWARE.

external file attributes: (4 bytes)

The mapping of the external attributes is host-system dependent (see 'version made by'). For MS-DOS, the low order byte is the MS-DOS directory attribute byte. If input came from standard input, this field is set to zero.

relative offset of local header: (4 bytes)

This is the offset from the start of the first disk on which this file appears, to where the local header should be found. If an archive is in zip64 format and the value in this field is 0xFFFFFFFF, the size will be in the corresponding 8 byte zip64 extended information extra field.

file name: (Variable)

    The name of the file, with optional relative path.
    The path stored should not contain a drive or
    device letter, or a leading slash. All slashes
    should be forward slashes '/' as opposed to
    backwards slashes '\' for compatibility with Amiga
    and Unix file systems etc. If input came from standard
    input, there is no file name field.

extra field: (Variable)

    This is for expansion. If additional information
    needs to be stored for special needs or for specific
    platforms, it should be stored here. Earlier versions
    of the software can then safely skip this file, and
    find the next file or header. This field will be 0
    length in version 1.0.

    In order to allow different programs and different types
    of information to be stored in the 'extra' field in .ZIP
    files, the following structure should be used for all
    programs storing data in this field:

    header1+data1 + header2+data2 . . .

    Each header should consist of:

      Header ID - 2 bytes
      Data Size - 2 bytes

    Note: all fields stored in Intel low-byte/high-byte order.

    The Header ID field indicates the type of data that is in
    the following data block.

    Header ID's of 0 thru 31 are reserved for use by PKWARE.
    The remaining ID's can be used by third party vendors for
    proprietary usage.

    The current Header ID mappings defined by PKWARE are:

    0x0001   ZIP64 extended information extra field
    0x0007   AV Info
    0x0009   OS/2
    0x000a   NTFS
    0x000c   OpenVMS
    0x000d   Unix
    0x000f   Patch Descriptor
    0x0014   PKCS#7 Store for X.509 Certificates
    0x0015   X.509 Certificate ID and Signature for
                  individual file
    0x0016   X.509 Certificate ID for Central Directory

    Third party mappings commonly used are:

    0x0065   IBM S/390 attributes - uncompressed
    0x0066   IBM S/390 attributes - compressed
    0x07c8   Macintosh
    0x2605   ZipIt Macintosh
    0x2705   ZIpIt Macintosh 1.3.5+
    0x334d   Info-ZIP Macintosh
    0x4341   Acorn/SparkFS
    0x4453   Windows NT security descriptor (binary ACL)
    0x4704   VM/CMS
    0x470f   MVS
    0x4b46   FWKCS MD5 (see below)
    0x4c41   OS/2 access control list (text ACL)
    0x4d49   Info-ZIP OpenVMS
    0x4f4c   Xceed original location extra field
    0x5356   AOS/VS (ACL)
    0x5455   extended timestamp
    0x5855   Info-ZIP Unix (original, also OS/2, NT, etc)
    0x554e   Xceed unicode extra field
    0x6542   BeOS/BeBox

0x756e	ASi Unix
0x7855	Info-ZIP Unix (new)
0xfd4a	SMS/QDOS

The Data Size field indicates the size of the following data block. Programs can use this value to skip to the next header block, passing over any data blocks that are not of interest.

Note: As stated above, the size of the entire .ZIP file header, including the file name, comment, and extra field should not exceed 64K in size.

In case two different programs should appropriate the same Header ID value, it is strongly recommended that each program place a unique signature of at least two bytes in size (and preferably 4 bytes or bigger) at the start of each data area. Every program should verify that its unique signature is present, in addition to the Header ID value being correct, before assuming that it is a block of known type.

-OS/2 Extra Field:

The following is the layout of the OS/2 attributes "extra" block. (Last Revision 09/05/95)

Note: all fields stored in Intel low-byte/high-byte order.

	Value	Size	Description
(OS/2)	0x0009	2 bytes	Tag for this "extra" block type
	TSize	2 bytes	Size for the following data block
	BSize	4 bytes	Uncompressed Block Size
	CType	2 bytes	Compression type
	EACRC	4 bytes	CRC value for uncompress block
	(var)	variable	Compressed block

The OS/2 extended attribute structure (FEA2LIST) is compressed and then stored in it's entirety within this structure. There will only ever be one "block" of data in VarFields[].

-UNIX Extra Field:

The following is the layout of the Unix "extra" block. Note: all fields are stored in Intel low-byte/high-byte order.

	Value	Size	Description
(UNIX)	0x000d	2 bytes	Tag for this "extra" block type
	TSize	2 bytes	Size for the following data block
	Atime	4 bytes	File last access time
	Mtime	4 bytes	File last modification time
	Uid	2 bytes	File user ID
	Gid	2 bytes	File group ID
	(var)	variable	Variable length data field

The variable length data field will contain file type specific data. Currently the only values allowed are the original "linked to" file names for hard or symbolic links, and the major and minor device node numbers for character and block device nodes. Since device nodes cannot be either symbolic or hard links, only one set of variable length data is stored. Link files will have the name of the original file stored. This name is NOT NULL terminated. Its size can be determined by checking TSize - 12. Device entries will have eight bytes stored as two 4 byte entries (in little endian format). The first entry will be the major device number, and the second the minor device number.

```
      -OpenVMS Extra Field:

       The following is the layout of the OpenVMS attributes 
       "extra" block.

       Note: all fields stored in Intel low-byte/high-byte order.

       Value      Size       Description
       -----      ----       -----------
(VMS)  0x000c     2 bytes    Tag for this "extra" block type
       TSize      2 bytes    Size of the total "extra" block
       CRC        4 bytes    32-bit CRC for remainder of the block
       Tag1       2 bytes    OpenVMS attribute tag value #1
       Size1      2 bytes    Size of attribute #1, in bytes
       (var.)     Size1      Attribute #1 data
       .
       .
       .
       TagN       2 bytes    OpenVMS attribute tage value #N
       SizeN      2 bytes    Size of attribute #N, in bytes
       (var.)     SizeN      Attribute #N data

       Rules:

       1. There will be one or more of attributes present, which 
          will each be preceded by the above TagX & SizeX values.  
          These values are identical to the ATR$C_XXXX and 
          ATR$S_XXXX constants which are defined in ATR.H under 
          OpenVMS C.  Neither of these values will ever be zero.

       2. No word alignment or padding is performed.

       3. A well-behaved PKZIP/OpenVMS program should never produce
          more than one sub-block with the same TagX value.  Also,
          there will never be more than one "extra" block of type
          0x000c in a particular directory record.

      -NTFS Extra Field:

       The following is the layout of the NTFS attributes 
       "extra" block. (Note: At this time the Mtime, Atime
       and Ctime values may be used on any WIN32 system.)  

       Note: all fields stored in Intel low-byte/high-byte order.

       Value      Size       Description
       -----      ----       -----------
(NTFS) 0x000a     2 bytes    Tag for this "extra" block type
       TSize      2 bytes    Size of the total "extra" block
       Reserved   4 bytes    Reserved for future use
       Tag1       2 bytes    NTFS attribute tag value #1
       Size1      2 bytes    Size of attribute #1, in bytes
       (var.)     Size1      Attribute #1 data
       .
       .
       .
       TagN       2 bytes    NTFS attribute tag value #N
       SizeN      2 bytes    Size of attribute #N, in bytes
       (var.)     SizeN      Attribute #N data

       For NTFS, values for Tag1 through TagN are as follows:
       (currently only one set of attributes is defined for NTFS)

       Tag        Size       Description
       -----      ----       -----------
       0x0001     2 bytes    Tag for attribute #1 
       Size1      2 bytes    Size of attribute #1, in bytes
       Mtime      8 bytes    File last modification time
       Atime      8 bytes    File last access time
       Ctime      8 bytes    File creation time

      -PATCH Descriptor Extra Field:
```

```
        The following is the layout of the Patch Descriptor "extra"
        block.

        Note: all fields stored in Intel low-byte/high-byte order.

        Value     Size     Description
        -----     ----     -----------
(Patch) 0x000f    2 bytes  Tag for this "extra" block type
        TSize     2 bytes  Size of the total "extra" block
        Version   2 bytes  Version of the descriptor
        Flags     4 bytes  Actions and reactions (see below)
        OldSize   4 bytes  Size of the file about to be patched
        OldCRC    4 bytes  32-bit CRC of the file to be patched
        NewSize   4 bytes  Size of the resulting file
        NewCRC    4 bytes  32-bit CRC of the resulting file

        Actions and reactions

        Bits          Description
        ----          ----------------
        0             Use for autodetection
        1             Treat as selfpatch
        2-3           RESERVED
        4-5           Action (see below)
        6-7           RESERVED
        8-9           Reaction (see below) to absent file
        10-11         Reaction (see below) to newer file
        12-13         Reaction (see below) to unknown file
        14-15         RESERVED
        16-31         RESERVED

        Actions

        Action       Value
        ------       -----
        none         0
        add          1
        delete       2
        patch        3

        Reactions

        Reaction     Value
        --------     -----
        ask          0
        skip         1
        ignore       2
        fail         3

       -PKCS#7 Store for X.509 Certificates

        Note: all fields stored in Intel low-byte/high-byte order.

        Value     Size     Description
        -----     ----     -----------
(Store) 0x0014    2 bytes  Tag for this "extra" block type
        SSize     2 bytes  Size of the store data
        SData   (variable) Data about the store

        SData
        Value     Size     Description
        -----     ----     -----------
        Version   2 bytes  Version number
        StoreD  (variable) Actual store data

         -X.509 Certificate ID and Signature for individual file

        Note: all fields stored in Intel low-byte/high-byte order.

        Value     Size     Description
        -----     ----     -----------
(CID)   0x0015    2 bytes  Tag for this "extra" block type
```

```
        CSize     2 bytes  Size of Method
        Method  (variable)

        Method
        Value     Size     Description
        -----     ----     -----------
        Version   2 bytes  Version number
        AlgID     2 bytes  Algorithm ID used for signing
        IDSize    2 bytes  Size of Certificate ID data
        CertID  (variable) Certificate ID data
        SigSize   2 bytes  Size of Signature data
        Sig     (variable) Signature data

        CertID
        Value     Size     Description
        -----     ----     -----------
        Size1     4 bytes  Size of CertID
        Size1     4 bytes  A bug in version one causes this value
                           to appear twice.
        IssSize   4 bytes  Issuer data size
        Issuer  (variable) Issuer data
        SerSize   4 bytes  Serial Number size
        Serial  (variable) Serial Number data

         -X.509 Certificate ID and Signature for central directory

        Note: all fields stored in Intel low-byte/high-byte order.

        Value     Size     Description
        -----     ----     -----------
(CDID)  0x0016    2 bytes  Tag for this "extra" block type
        CSize     2 bytes  Size of Method
        Method  (variable)


       -ZIP64 Extended Information Extra Field:

        The following is the layout of the ZIP64 extended
        information "extra" block. If one of the size or
        offset fields in the Local or Central directory
        record is too small to hold the required data,
        a ZIP64 extended information record is created.
        The order of the fields in the ZIP64 extended
        information record is fixed, but the fields will
        only appear if the corresponding Local or Central
        directory record field is set to 0xFFFF or 0xFFFFFFFF.

        Note: all fields stored in Intel low-byte/high-byte order.

        Value      Size       Description
        -----      ----       -----------
(ZIP64) 0x0001     2 bytes    Tag for this "extra" block type
        Size       2 bytes    Size of this "extra" block
        Original
        Size       8 bytes    Original uncompresseed file size
        Compressed
        Size       8 bytes    Size of compressed data
        Relative Header
        Offset     8 bytes    Offset of local header record
        Disk Start
        Number     4 bytes    Number of the disk on which
                              this file starts

        This entry in the Local header must include BOTH original
        and compressed file sizes.

       - FWKCS MD5 Extra Field:

        The FWKCS Contents_Signature System, used in
        automatically identifying files independent of file name,
        optionally adds and uses an extra field to support the
        rapid creation of an enhanced contents_signature:
```

```
        Header ID = 0x4b46
        Data Size = 0x0013
        Preface   = 'M','D','5'
        followed by 16 bytes containing the uncompressed file's
        128_bit MD5 hash(1), low byte first.

    When FWKCS revises a .ZIP file central directory to add
    this extra field for a file, it also replaces the
    central directory entry for that file's uncompressed
    file length with a measured value.

    FWKCS provides an option to strip this extra field, if
    present, from a .ZIP file central directory. In adding
    this extra field, FWKCS preserves .ZIP file Authenticity
    Verification; if stripping this extra field, FWKCS
    preserves all versions of AV through PKZIP version 2.04g.

    FWKCS, and FWKCS Contents_Signature System, are
    trademarks of Frederick W. Kantor.

    (1) R. Rivest, RFC1321.TXT, MIT Laboratory for Computer
        Science and RSA Data Security, Inc., April 1992.
        ll.76-77: "The MD5 algorithm is being placed in the
        public domain for review and possible adoption as a
        standard."

file comment: (Variable)

    The comment for this file.

number of this disk: (2 bytes)

    The number of this disk, which contains central
    directory end record. If an archive is in zip64 format
    and the value in this field is 0xFFFF, the size will
    be in the corresponding 4 byte zip64 end of central
    directory field.


number of the disk with the start of the central
directory: (2 bytes)

    The number of the disk on which the central
    directory starts. If an archive is in zip64 format
    and the value in this field is 0xFFFF, the size will
    be in the corresponding 4 byte zip64 end of central
    directory field.

total number of entries in the central dir on
this disk: (2 bytes)

    The number of central directory entries on this disk.
    If an archive is in zip64 format and the value in
    this field is 0xFFFF, the size will be in the
    corresponding 8 byte zip64 end of central
    directory field.

total number of entries in the central dir: (2 bytes)

    The total number of files in the .ZIP file. If an
    archive is in zip64 format and the value in this field
    is 0xFFFF, the size will be in the corresponding 8 byte
    zip64 end of central directory field.

size of the central directory: (4 bytes)

    The size (in bytes) of the entire central directory.
    If an archive is in zip64 format and the value in
    this field is 0xFFFFFFFF, the size will be in the
    corresponding 8 byte zip64 end of central
    directory field.

offset of start of central directory with respect to
```

the starting disk number: (4 bytes)

Offset of the start of the central directory on the disk on which the central directory starts. If an archive is in zip64 format and the value in this field is 0xFFFFFFFF, the size will be in the corresponding 8 byte zip64 end of central directory field.

.ZIP file comment length: (2 bytes)

The length of the comment for this .ZIP file.

.ZIP file comment: (Variable)

The comment for this .ZIP file.

zip64 extensible data sector   (variable size)

(currently reserved for use by PKWARE)

I. General notes:

1) All fields unless otherwise noted are unsigned and stored in Intel low-byte:high-byte, low-word:high-word order.

2) String fields are not null terminated, since the length is given explicitly.

3) Local headers should not span disk boundaries. Also, even though the central directory can span disk boundaries, no single record in the central directory should be split across disks.

4) The entries in the central directory may not necessarily be in the same order that files appear in the .ZIP file.

5) Spanned/Split archives created using PKZIP for Windows (V2.50 or greater), PKZIP Command Line (V2.50 or greater), or PKZIP Explorer will include a special spanning signature as the first 4 bytes of the first segment of the archive. This signature (0x08074b50) will be followed immediately by the local header signature for the first file in the archive. A special spanning marker may also appear in spanned/split archives if the spanning or splitting process starts but only requires one segement. In this case the 0x08074b50 signature will be replaced with the temporary spanning marker signature of 0x30304b50. Spanned/split archives created with this special signature are compatible with all versions of PKZIP from PKWARE. Split archives can only be uncompressed by other versions of PKZIP that know how to create a split archive.

6) If one of the fields in the end of central directory record is too small to hold required data, the field should be set to -1 (0xFFFF or 0xFFFFFFFF) and the Zip64 format record should be created.

7) The end of central directory record and the Zip64 end of central directory locator record must reside on the same disk when splitting or spanning an archive.

UnShrinking - Method 1

Shrinking is a Dynamic Ziv-Lempel-Welch compression algorithm with partial clearing. The initial code size is 9 bits, and the maximum code size is 13 bits. Shrinking differs from conventional Dynamic Ziv-Lempel-Welch implementations in several respects:

1) The code size is controlled by the compressor, and is not automatically increased when codes larger than the current code size are created (but not necessarily used). When the decompressor encounters the code sequence 256 (decimal) followed by 1, it should increase the code size read from the input stream to the next bit size. No blocking of the codes is performed, so the next code at the increased size should be read from the input stream immediately after where the previous code at the smaller bit size was read. Again, the decompressor should not increase the code size used until the sequence 256,1 is encountered.

2) When the table becomes full, total clearing is not performed. Rather, when the compressor emits the code sequence 256,2 (decimal), the decompressor should clear all leaf nodes from the Ziv-Lempel tree, and continue to use the current code size. The nodes that are cleared from the Ziv-Lempel tree are then re-used, with the lowest code value re-used first, and the highest code value re-used last. The compressor can emit the sequence 256,2 at any time.

Expanding - Methods 2-5

The Reducing algorithm is actually a combination of two distinct algorithms. The first algorithm compresses repeated byte sequences, and the second algorithm takes the compressed stream from the first algorithm and applies a probabilistic compression method.

The probabilistic compression stores an array of 'follower sets' S(j), for j=0 to 255, corresponding to each possible ASCII character. Each set contains between 0 and 32 characters, to be denoted as S(j)[0],...,S(j)[m], where m<32. The sets are stored at the beginning of the data area for a Reduced file, in reverse order, with S(255) first, and S(0) last.

The sets are encoded as { N(j), S(j)[0],...,S(j)[N(j)-1] }, where N(j) is the size of set S(j). N(j) can be 0, in which case the follower set for S(j) is empty. Each N(j) value is encoded in 6 bits, followed by N(j) eight bit character values corresponding to S(j)[0] to S(j)[N(j)-1] respectively. If N(j) is 0, then no values for S(j) are stored, and the value for N(j-1) immediately follows.

Immediately after the follower sets, is the compressed data stream. The compressed data stream can be interpreted for the probabilistic decompression as follows:

```
let Last-Character <- 0.
loop until done
    if the follower set S(Last-Character) is empty then
        read 8 bits from the input stream, and copy this
        value to the output stream.
    otherwise if the follower set S(Last-Character) is non-empty then
        read 1 bit from the input stream.
        if this bit is not zero then
            read 8 bits from the input stream, and copy this
            value to the output stream.
        otherwise if this bit is zero then
            read B(N(Last-Character)) bits from the input
            stream, and assign this value to I.
            Copy the value of S(Last-Character)[I] to the
            output stream.

    assign the last value placed on the output stream to
    Last-Character.
end loop
```

B(N(j)) is defined as the minimal number of bits required to
encode the value N(j)-1.

The decompressed stream from above can then be expanded to
re-create the original file as follows:

```
let State <- 0.

loop until done
    read 8 bits from the input stream into C.
    case State of
        0:  if C is not equal to DLE (144 decimal) then
                copy C to the output stream.
            otherwise if C is equal to DLE then
                let State <- 1.

        1:  if C is non-zero then
                let V <- C.
                let Len <- L(V)
                let State <- F(Len).
            otherwise if C is zero then
                copy the value 144 (decimal) to the output stream.
                let State <- 0

        2:  let Len <- Len + C
            let State <- 3.

        3:  move backwards D(V,C) bytes in the output stream
            (if this position is before the start of the output
            stream, then assume that all the data before the
            start of the output stream is filled with zeros).
            copy Len+3 bytes from this position to the output stream.
            let State <- 0.
    end case
end loop
```

The functions F,L, and D are dependent on the 'compression
factor', 1 through 4, and are defined as follows:

```
For compression factor 1:
    L(X) equals the lower 7 bits of X.
    F(X) equals 2 if X equals 127 otherwise F(X) equals 3.
    D(X,Y) equals the (upper 1 bit of X) * 256 + Y + 1.
For compression factor 2:
    L(X) equals the lower 6 bits of X.
    F(X) equals 2 if X equals 63 otherwise F(X) equals 3.
    D(X,Y) equals the (upper 2 bits of X) * 256 + Y + 1.
For compression factor 3:
    L(X) equals the lower 5 bits of X.
    F(X) equals 2 if X equals 31 otherwise F(X) equals 3.
    D(X,Y) equals the (upper 3 bits of X) * 256 + Y + 1.
For compression factor 4:
    L(X) equals the lower 4 bits of X.
    F(X) equals 2 if X equals 15 otherwise F(X) equals 3.
    D(X,Y) equals the (upper 4 bits of X) * 256 + Y + 1.
```

Imploding - Method 6

The Imploding algorithm is actually a combination of two distinct
algorithms. The first algorithm compresses repeated byte
sequences using a sliding dictionary. The second algorithm is
used to compress the encoding of the sliding dictionary output,
using multiple Shannon-Fano trees.

The Imploding algorithm can use a 4K or 8K sliding dictionary
size. The dictionary size used can be determined by bit 1 in the
general purpose flag word; a 0 bit indicates a 4K dictionary
while a 1 bit indicates an 8K dictionary.

The Shannon-Fano trees are stored at the start of the compressed
file. The number of trees stored is defined by bit 2 in the
general purpose flag word; a 0 bit indicates two trees stored, a

1 bit indicates three trees are stored. If 3 trees are stored, the first Shannon-Fano tree represents the encoding of the Literal characters, the second tree represents the encoding of the Length information, the third represents the encoding of the Distance information. When 2 Shannon-Fano trees are stored, the Length tree is stored first, followed by the Distance tree.

The Literal Shannon-Fano tree, if present is used to represent the entire ASCII character set, and contains 256 values. This tree is used to compress any data not compressed by the sliding dictionary algorithm. When this tree is present, the Minimum Match Length for the sliding dictionary is 3. If this tree is not present, the Minimum Match Length is 2.

The Length Shannon-Fano tree is used to compress the Length part of the (length,distance) pairs from the sliding dictionary output. The Length tree contains 64 values, ranging from the Minimum Match Length, to 63 plus the Minimum Match Length.

The Distance Shannon-Fano tree is used to compress the Distance part of the (length,distance) pairs from the sliding dictionary output. The Distance tree contains 64 values, ranging from 0 to 63, representing the upper 6 bits of the distance value. The distance values themselves will be between 0 and the sliding dictionary size, either 4K or 8K.

The Shannon-Fano trees themselves are stored in a compressed format. The first byte of the tree data represents the number of bytes of data representing the (compressed) Shannon-Fano tree minus 1. The remaining bytes represent the Shannon-Fano tree data encoded as:

```
    High 4 bits: Number of values at this bit length + 1. (1 - 16)
    Low  4 bits: Bit Length needed to represent value + 1. (1 - 16)
```

The Shannon-Fano codes can be constructed from the bit lengths using the following algorithm:

1) Sort the Bit Lengths in ascending order, while retaining the order of the original lengths stored in the file.

2) Generate the Shannon-Fano trees:

```
    Code <- 0
    CodeIncrement <- 0
    LastBitLength <- 0
    i <- number of Shannon-Fano codes - 1   (either 255 or 63)

    loop while i >= 0
        Code = Code + CodeIncrement
        if BitLength(i) <> LastBitLength then
            LastBitLength=BitLength(i)
            CodeIncrement = 1 shifted left (16 - LastBitLength)
        ShannonCode(i) = Code
        i <- i - 1
    end loop
```

3) Reverse the order of all the bits in the above ShannonCode() vector, so that the most significant bit becomes the least significant bit. For example, the value 0x1234 (hex) would become 0x2C48 (hex).

4) Restore the order of Shannon-Fano codes as originally stored within the file.

Example:

This example will show the encoding of a Shannon-Fano tree of size 8. Notice that the actual Shannon-Fano trees used for Imploding are either 64 or 256 entries in size.

Example: 0x02, 0x42, 0x01, 0x13

The first byte indicates 3 values in this table. Decoding the bytes:

    0x42 = 5 codes of 3 bits long
    0x01 = 1 code of 2 bits long
    0x13 = 2 codes of 4 bits long

This would generate the original bit length array of:
(3, 3, 3, 3, 3, 2, 4, 4)

There are 8 codes in this table for the values 0 thru 7. Using the algorithm to obtain the Shannon-Fano codes produces:

Val	Sorted	Constructed Code	Reversed Value	Order Restored	Original Length
0:	2	1100000000000000	11	101	3
1:	3	1010000000000000	101	001	3
2:	3	1000000000000000	001	110	3
3:	3	0110000000000000	110	010	3
4:	3	0100000000000000	010	100	3
5:	3	0010000000000000	100	11	2
6:	4	0001000000000000	1000	1000	4
7:	4	0000000000000000	0000	0000	4

The values in the Val, Order Restored and Original Length columns now represent the Shannon-Fano encoding tree that can be used for decoding the Shannon-Fano encoded data. How to parse the variable length Shannon-Fano values from the data stream is beyond the scope of this document. (See the references listed at the end of this document for more information.) However, traditional decoding schemes used for Huffman variable length decoding, such as the Greenlaw algorithm, can be successfully applied.

The compressed data stream begins immediately after the compressed Shannon-Fano data. The compressed data stream can be interpreted as follows:

```
loop until done
    read 1 bit from input stream.

    if this bit is non-zero then       (encoded data is literal data)
        if Literal Shannon-Fano tree is present
            read and decode character using Literal Shannon-Fano tree.
        otherwise
            read 8 bits from input stream.
        copy character to the output stream.
    otherwise              (encoded data is sliding dictionary match)
        if 8K dictionary size
            read 7 bits for offset Distance (lower 7 bits of offset).
        otherwise
            read 6 bits for offset Distance (lower 6 bits of offset).

        using the Distance Shannon-Fano tree, read and decode the
          upper 6 bits of the Distance value.

        using the Length Shannon-Fano tree, read and decode
          the Length value.

        Length <- Length + Minimum Match Length

        if Length = 63 + Minimum Match Length
            read 8 bits from the input stream,
            add this value to Length.

        move backwards Distance+1 bytes in the output stream, and
        copy Length characters from this position to the output
        stream.  (if this position is before the start of the output
        stream, then assume that all the data before the start of
        the output stream is filled with zeros).
end loop
```

Tokenizing - Method 7

This method is not used by PKZIP.

Deflating - Method 8

The Deflate algorithm is similar to the Implode algorithm using
a sliding dictionary of up to 32K with secondary compression
from Huffman/Shannon-Fano codes.

The compressed data is stored in blocks with a header describing
the block and the Huffman codes used in the data block. The header
format is as follows:

```
   Bit 0: Last Block bit     This bit is set to 1 if this is the last
                             compressed block in the data.
   Bits 1-2: Block type
      00 (0) - Block is stored - All stored data is byte aligned.
               Skip bits until next byte, then next word = block
               length, followed by the ones compliment of the block
               length word. Remaining data in block is the stored
               data.

      01 (1) - Use fixed Huffman codes for literal and distance codes.
               Lit Code    Bits             Dist Code   Bits
               ---------   ----             ---------   ----
                 0 - 143    8                 0 - 31      5
               144 - 255    9
               256 - 279    7
               280 - 287    8

               Literal codes 286-287 and distance codes 30-31 are
               never used but participate in the huffman construction.

      10 (2) - Dynamic Huffman codes.  (See expanding Huffman codes)

      11 (3) - Reserved - Flag a "Error in compressed data" if seen.
```

Expanding Huffman Codes

If the data block is stored with dynamic Huffman codes, the Huffman
codes are sent in the following compressed format:

```
   5 Bits: # of Literal codes sent - 256 (256 - 286)
           All other codes are never sent.
   5 Bits: # of Dist codes - 1           (1 - 32)
   4 Bits: # of Bit Length codes - 3     (3 - 19)
```

The Huffman codes are sent as bit lengths and the codes are built as
described in the implode algorithm. The bit lengths themselves are
compressed with Huffman codes. There are 19 bit length codes:

```
   0 - 15: Represent bit lengths of 0 - 15
       16: Copy the previous bit length 3 - 6 times.
           The next 2 bits indicate repeat length (0 = 3, ... ,3 = 6)
              Example:  Codes 8, 16 (+2 bits 11), 16 (+2 bits 10) will
                        expand to 12 bit lengths of 8 (1 + 6 + 5)
       17: Repeat a bit length of 0 for 3 - 10 times. (3 bits of length)
       18: Repeat a bit length of 0 for 11 - 138 times (7 bits of length)
```

The lengths of the bit length codes are sent packed 3 bits per value
(0 - 7) in the following order:

```
   16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15
```

The Huffman codes should be built as described in the Implode algorithm
except codes are assigned starting at the shortest bit length, i.e. the
shortest code should be all 0's rather than all 1's. Also, codes with
a bit length of zero do not participate in the tree construction. The
codes are then used to decode the bit lengths for the literal and
distance tables.

The bit lengths for the literal tables are sent first with the number

of entries sent described by the 5 bits sent earlier. There are up to 286 literal characters; the first 256 represent the respective 8 bit character, code 256 represents the End-Of-Block code, the remaining 29 codes represent copy lengths of 3 thru 258. There are up to 30 distance codes representing distances from 1 thru 32k as described below.

Length Codes

Code	Extra Bits	Length	Code	Extra Bits	Lengths	Code	Extra Bits	Lengths	Code	Extra Bits	Length(s)
257	0	3	265	1	11,12	273	3	35-42	281	5	131-162
258	0	4	266	1	13,14	274	3	43-50	282	5	163-194
259	0	5	267	1	15,16	275	3	51-58	283	5	195-226
260	0	6	268	1	17,18	276	3	59-66	284	5	227-257
261	0	7	269	2	19-22	277	4	67-82	285	0	258
262	0	8	270	2	23-26	278	4	83-98			
263	0	9	271	2	27-30	279	4	99-114			
264	0	10	272	2	31-34	280	4	115-130			

Distance Codes

Code	Extra Bits	Dist	Code	Extra Bits	Dist	Code	Extra Bits	Distance	Code	Extra Bits	Distance
0	0	1	8	3	17-24	16	7	257-384	24	11	4097-6144
1	0	2	9	3	25-32	17	7	385-512	25	11	6145-8192
2	0	3	10	4	33-48	18	8	513-768	26	12	8193-12288
3	0	4	11	4	49-64	19	8	769-1024	27	12	12289-16384
4	1	5,6	12	5	65-96	20	9	1025-1536	28	13	16385-24576
5	1	7,8	13	5	97-128	21	9	1537-2048	29	13	24577-32768
6	2	9-12	14	6	129-192	22	10	2049-3072			
7	2	13-16	15	6	193-256	23	10	3073-4096			

The compressed data stream begins immediately after the compressed header data. The compressed data stream can be interpreted as follows:

```
do
   read header from input stream.

   if stored block
      skip bits until byte aligned
      read count and 1's compliment of count
      copy count bytes data block
   otherwise
      loop until end of block code sent
         decode literal character from input stream
         if literal < 256
            copy character to the output stream
         otherwise
            if literal = end of block
               break from loop
            otherwise
               decode distance from input stream

               move backwards distance bytes in the output stream, and
               copy length characters from this position to the output
               stream.
      end loop
while not last block

if data descriptor exists
   skip bits until byte aligned
   read crc and sizes
endif
```

Decryption

The encryption used in PKZIP was generously supplied by Roger Schlafly. PKWARE is grateful to Mr. Schlafly for his expert

help and advice in the field of data encryption.

PKZIP encrypts the compressed data stream. Encrypted files must be decrypted before they can be extracted.

Each encrypted file has an extra 12 bytes stored at the start of the data area defining the encryption header for that file. The encryption header is originally set to random values, and then itself encrypted, using three, 32-bit keys. The key values are initialized using the supplied encryption password. After each byte is encrypted, the keys are then updated using pseudo-random number generation techniques in combination with the same CRC-32 algorithm used in PKZIP and described elsewhere in this document.

The following is the basic steps required to decrypt a file:

1) Initialize the three 32-bit keys with the password.
2) Read and decrypt the 12-byte encryption header, further initializing the encryption keys.
3) Read and decrypt the compressed data stream using the encryption keys.

Step 1 - Initializing the encryption keys

```
Key(0) <- 305419896
Key(1) <- 591751049
Key(2) <- 878082192

loop for i <- 0 to length(password)-1
    update_keys(password(i))
end loop
```

Where update_keys() is defined as:

```
update_keys(char):
  Key(0) <- crc32(key(0),char)
  Key(1) <- Key(1) + (Key(0) & 000000ffH)
  Key(1) <- Key(1) * 134775813 + 1
  Key(2) <- crc32(key(2),key(1) >> 24)
end update_keys
```

Where crc32(old_crc,char) is a routine that given a CRC value and a character, returns an updated CRC value after applying the CRC-32 algorithm described elsewhere in this document.

Step 2 - Decrypting the encryption header

The purpose of this step is to further initialize the encryption keys, based on random data, to render a plaintext attack on the data ineffective.

Read the 12-byte encryption header into Buffer, in locations Buffer(0) thru Buffer(11).

```
loop for i <- 0 to 11
    C <- buffer(i) ^ decrypt_byte()
    update_keys(C)
    buffer(i) <- C
end loop
```

Where decrypt_byte() is defined as:

```
unsigned char decrypt_byte()
    local unsigned short temp
    temp <- Key(2) | 2
    decrypt_byte <- (temp * (temp ^ 1)) >> 8
end decrypt_byte
```

After the header is decrypted, the last 1 or 2 bytes in Buffer should be the high-order word/byte of the CRC for the file being decrypted, stored in Intel low-byte/high-byte order. Versions of

PKZIP prior to 2.0 used a 2 byte CRC check; a 1 byte CRC check is
used on versions after 2.0. This can be used to test if the password
supplied is correct or not.

Step 3 - Decrypting the compressed data stream

The compressed data stream can be decrypted as follows:

```
loop until done
    read a character into C
    Temp <- C ^ decrypt_byte()
    update_keys(temp)
    output Temp
end loop
```

Change Process

In order for the .ZIP file format to remain a viable definition, this
specification should be considered as open for periodic review and
revision. Although this format was originally designed with a
certain level of extensibility, not all changes in technology
(present or future) were or will be necessarily considered in its
design. If your application requires new definitions to the
extensible sections in this format, or if you would like to
submit new data structures, please forward your request to
zipformat@pkware.com. All submissions will be reviewed by the
ZIP File Specification Committee for possible inclusion into
future versions of this specification. Periodic revisions
to this specification will be published to ensure interoperability.

Acknowledgements

In addition to the above mentioned contributors to PKZIP and PKUNZIP,
I would like to extend special thanks to Robert Mahoney for suggesting
the extension .ZIP for this software.

References:

Fiala, Edward R., and Greene, Daniel H., "Data compression with finite windows", Communications of the ACM, Volume 32, Number 4, April 1989, pages 490-505.

Held, Gilbert, "Data Compression, Techniques and Applications, Hardware and Software Considerations", John Wiley & Sons, 1987.

Huffman, D.A., "A method for the construction of minimum-redundancy codes", Proceedings of the IRE, Volume 40, Number 9, September 1952, pages 1098-1101.

Nelson, Mark, "LZW Data Compression", Dr. Dobbs Journal, Volume 14, Number 10, October 1989, pages 29-37.

Nelson, Mark, "The Data Compression Book", M&T Books, 1991.

Storer, James A., "Data Compression, Methods and Theory", Computer Science Press, 1988

Welch, Terry, "A Technique for High-Performance Data Compression", IEEE Computer, Volume 17, Number 6, June 1984, pages 8-19.

Ziv, J. and Lempel, A., "A universal algorithm for sequential data compression", Communications of the ACM, Volume 30, Number 6, June 1987, pages 520-540.

Ziv, J. and Lempel, A., "Compression of individual sequences via variable-rate coding", IEEE Transactions on Information Theory, Volume 24, Number 5, September 1978, pages 530-536.