

Held Over by Popular Demand!

Tutorial on MIDI and Music Synthesis

**Written by Jim Heckroth, Crystal Semiconductor Corp.
Used with Permission.**

**Published by:
The MIDI Manufacturers Association
POB 3173
La Habra CA 90632-3173**

Windows is a trademark of Microsoft Corporation. MPU-401, MT-32, LAPC-1 and Sound Canvas are trademarks of Roland Corporation. Sound Blaster is a trademark of Creative Labs, Inc. All other brand or product names mentioned are trademarks or registered trademarks of their respective holders.

**Copyright 1995 MIDI Manufacturers Association. All rights reserved.
No part of this document may be reproduced or copied without written permission of the publisher.**

Printed 1995
HTML coding by Scott Lehman

Table of Contents

- Introduction
- MIDI vs. Digitized Audio
- MIDI Basics
- MIDI Messages
- MIDI Sequencers and Standard MIDI Files
- Synthesizer Basics
- The General MIDI (GM) System
- Synthesis Technology: FM and Wavetable
- The PC to MIDI Connection
- Multimedia PC (MPC) Systems
- Microsoft Windows Configuration
- Summary

Introduction

The Musical Instrument Digital Interface (MIDI) protocol has been widely accepted and utilized by musicians and composers since its conception in the 1982/1983 time frame. MIDI data is a very efficient

method of representing musical performance information, and this makes MIDI an attractive protocol not only for composers or performers, but also for computer applications which produce sound, such as multimedia presentations or computer games. However, the lack of standardization of synthesizer capabilities hindered applications developers and presented new MIDI users with a rather steep learning curve to overcome.

Fortunately, thanks to the publication of the General MIDI System specification, wide acceptance of the most common PC/MIDI interfaces, support for MIDI in Microsoft WINDOWS and other operating systems, and the evolution of low-cost music synthesizers, the MIDI protocol is now seeing widespread use in a growing number of applications. This document is an overview of the standards, practices and terminology associated with the generation of sound using the MIDI protocol.

MIDI vs. Digitized Audio

Originally developed to allow musicians to connect synthesizers together, the MIDI protocol is now finding widespread use as a delivery medium to replace or supplement digitized audio in games and multimedia applications. There are several advantages to generating sound with a MIDI synthesizer rather than using sampled audio from disk or CD-ROM. The first advantage is storage space. Data files used to store digitally sampled audio in PCM format (such as .WAV files) tend to be quite large. This is especially true for lengthy musical pieces captured in stereo using high sampling rates.

MIDI data files, on the other hand, are extremely small when compared with sampled audio files. For instance, files containing high quality stereo sampled audio require about 10 Mbytes of data per minute of sound, while a typical MIDI sequence might consume less than 10 Kbytes of data per minute of sound. This is because the MIDI file does not contain the sampled audio data, it contains only the instructions needed by a synthesizer to play the sounds. These instructions are in the form of MIDI messages, which instruct the synthesizer which sounds to use, which notes to play, and how loud to play each note. The actual sounds are then generated by the synthesizer.

For computers, the smaller file size also means that less of the PC's bandwidth is utilized in spooling this data out to the peripheral which is generating sound. Other advantages of utilizing MIDI to generate sounds include the ability to easily edit the music, and the ability to change the playback speed and the pitch or key of the sounds independently. This last point is particularly important in synthesis applications such as karaoke equipment, where the musical key and tempo of a song may be selected by the user.

MIDI Basics

The Musical Instrument Digital Interface (MIDI) protocol provides a standardized and efficient means of conveying musical performance information as electronic data. MIDI information is transmitted in "MIDI messages", which can be thought of as instructions which tell a music synthesizer how to play a piece of music. The synthesizer receiving the MIDI data must generate the actual sounds. The MIDI 1.0 Detailed Specification provides a complete description of the MIDI protocol.

The MIDI data stream is a unidirectional asynchronous bit stream at 31.25 Kbits/sec. with 10 bits transmitted per byte (a start bit, 8 data bits, and one stop bit). The MIDI interface on a MIDI instrument will generally include three different MIDI connectors, labeled IN, OUT, and THRU. The MIDI data stream is usually originated by a MIDI controller, such as a musical instrument keyboard, or by a MIDI sequencer. A MIDI controller is a device which is played as an instrument, and it translates the performance into a MIDI data stream in real time (as it is played). A MIDI sequencer is a device which allows MIDI data sequences to be captured, stored, edited, combined, and replayed. The MIDI data output from a MIDI controller or sequencer is transmitted via the devices' MIDI OUT connector.

The recipient of this MIDI data stream is commonly a MIDI sound generator or sound module, which will receive MIDI messages at its MIDI IN connector, and respond to these messages by playing sounds. Figure 1 shows a simple MIDI system, consisting of a MIDI keyboard controller and a MIDI sound module. Note that many MIDI keyboard instruments include both the keyboard controller and the MIDI sound module functions within the same unit. In these units, there is an internal link between the keyboard and the sound module which may be enabled or disabled by setting the "local control" function of the instrument to ON or OFF respectively.

The single physical MIDI Channel is divided into 16 logical channels by the inclusion of a 4 bit Channel number within many of the MIDI messages. A musical instrument keyboard can generally be set to transmit on any one of the sixteen MIDI channels. A MIDI sound source, or sound module, can be set to receive on specific MIDI Channel(s). In the system depicted in Figure 1, the sound module would have to be set to receive the Channel which the keyboard controller is transmitting on in order to play sounds.

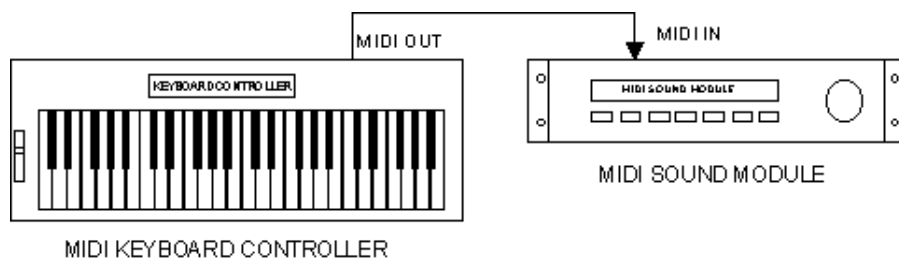


Figure 1: A Simple MIDI System

Information received on the MIDI IN connector of a MIDI device is transmitted back out (repeated) at the devices' MIDI THRU connector. Several MIDI sound modules can be daisy-chained by connecting the THRU output of one device to the IN connector of the next device downstream in the chain.

Figure 2 shows a more elaborate MIDI system. In this case, a MIDI keyboard controller is used as an input device to a MIDI sequencer, and there are several sound modules connected to the sequencer's MIDI OUT port. A composer might utilize a system like this to write a piece of music consisting of several different parts, where each part is written for a different instrument. The composer would play the individual parts on the keyboard one at a time, and these individual parts would be captured by the sequencer. The sequencer would then play the parts back together through the sound modules. Each part would be played on a different MIDI Channel, and the sound modules would be set to receive different channels. For example, Sound module number 1 might be set to play the part received on Channel 1 using a piano sound, while module 2 plays the information received on Channel 5 using an acoustic bass sound, and the drum machine plays the percussion part received on MIDI Channel 10.

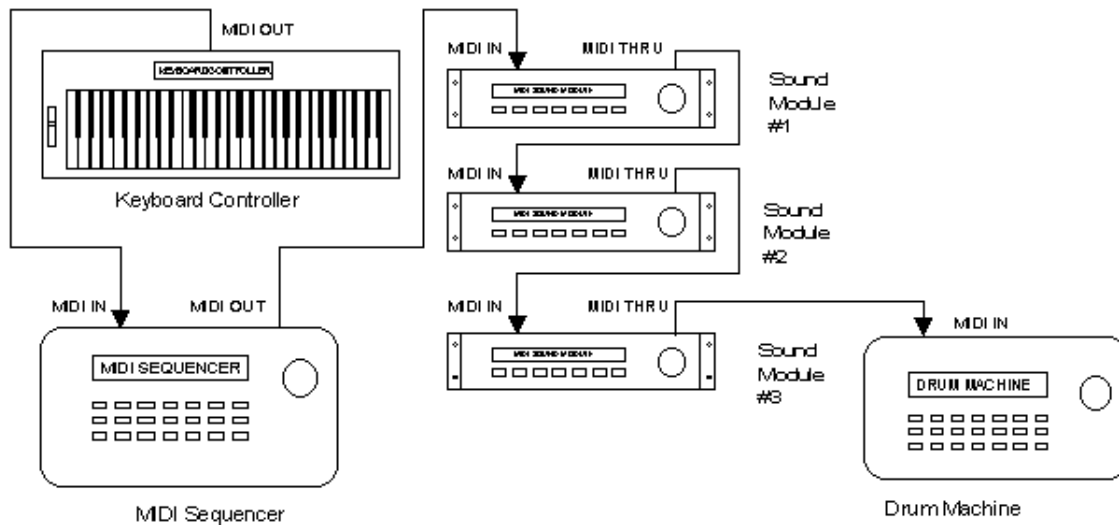


Figure 2: An Expanded MIDI System

In this example, a different sound module is used to play each part. However, sound modules which are "multitimbral" are capable of playing several different parts simultaneously. A single multitimbral sound module might be configured to receive the piano part on Channel 1, the bass part on Channel 5, and the drum part on Channel 10, and would play all three parts simultaneously.

Figure 3 depicts a PC-based MIDI system. In this system, the PC is equipped with an internal MIDI interface card which sends MIDI data to an external multitimbral MIDI synthesizer module. Application software, such as Multimedia presentation packages, educational software, or games, sends MIDI data to the MIDI interface card in parallel form over the PC bus. The MIDI interface converts this information into serial MIDI data which is sent to the sound module. Since this is a multitimbral module, it can play many different musical parts, such as piano, bass and drums, at the same time. Sophisticated MIDI sequencer software packages are also available for the PC. With this software running on the PC, a user could connect a MIDI keyboard controller to the MIDI IN port of the MIDI interface card, and have the same music composition capabilities discussed in the last two paragraphs.

There are a number of different configurations of PC-based MIDI systems possible. For instance, the MIDI interface and the MIDI sound module might be combined on the PC add-in card. In fact, the Multimedia PC (MPC) Specification requires that all MPC systems include a music synthesizer, and the synthesizer is normally included on the audio adapter card (the "sound card") along with the MIDI interface function. Until recently, most PC sound cards included FM synthesizers with limited capabilities and marginal sound quality. With these systems, an external wavetable synthesizer module might be added to get better sound quality. Recently, more advanced sound cards have been appearing which include high quality wavetable music synthesizers on-board, or as a daughter-card options. With the increasing use of the MIDI protocol in PC applications, this trend is sure to continue.

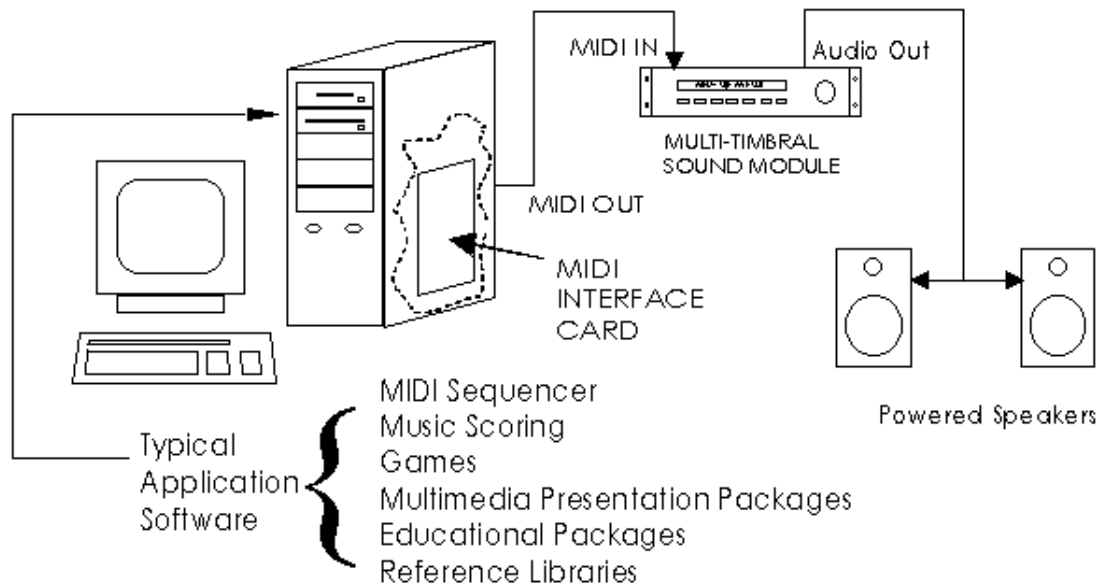


Figure 3: A PC-Based MIDI System

MIDI Messages

A MIDI message is made up of an eight-bit status byte which is generally followed by one or two data bytes. There are a number of different types of MIDI messages. At the highest level, MIDI messages are classified as being either Channel Messages or System Messages. Channel messages are those which apply to a specific Channel, and the Channel number is included in the status byte for these messages. System messages are not Channel specific, and no Channel number is indicated in their status bytes.

Channel Messages may be further classified as being either Channel Voice Messages, or Mode Messages. Channel Voice Messages carry musical performance data, and these messages comprise most of the traffic in a typical MIDI data stream. Channel Mode messages affect the way a receiving instrument will respond to the Channel Voice messages.

Channel Voice Messages

Channel Voice Messages are used to send musical performance information. The messages in this category are the Note On, Note Off, Polyphonic Key Pressure, Channel Pressure, Pitch Bend Change, Program Change, and the Control Change messages.

Note On / Note Off / Velocity

In MIDI systems, the activation of a particular note and the release of the same note are considered as two separate events. When a key is pressed on a MIDI keyboard instrument or MIDI keyboard controller, the keyboard sends a Note On message on the MIDI OUT port. The keyboard may be set to transmit on any one of the sixteen logical MIDI channels, and the status byte for the Note On message will indicate the selected Channel number. The Note On status byte is followed by two data bytes, which

specify key number (indicating which key was pressed) and velocity (how hard the key was pressed).

The key number is used in the receiving synthesizer to select which note should be played, and the velocity is normally used to control the amplitude of the note. When the key is released, the keyboard instrument or controller will send a Note Off message. The Note Off message also includes data bytes for the key number and for the velocity with which the key was released. The Note Off velocity information is normally ignored.

Aftertouch

Some MIDI keyboard instruments have the ability to sense the amount of pressure which is being applied to the keys while they are depressed. This pressure information, commonly called "aftertouch", may be used to control some aspects of the sound produced by the synthesizer (vibrato, for example). If the keyboard has a pressure sensor for each key, then the resulting "polyphonic aftertouch" information would be sent in the form of Polyphonic Key Pressure messages. These messages include separate data bytes for key number and pressure amount. It is currently more common for keyboard instruments to sense only a single pressure level for the entire keyboard. This "Channel aftertouch" information is sent using the Channel Pressure message, which needs only one data byte to specify the pressure value.

Pitch Bend

The Pitch Bend Change message is normally sent from a keyboard instrument in response to changes in position of the pitch bend wheel. The pitch bend information is used to modify the pitch of sounds being played on a given Channel. The Pitch Bend message includes two data bytes to specify the pitch bend value. Two bytes are required to allow fine enough resolution to make pitch changes resulting from movement of the pitch bend wheel seem to occur in a continuous manner rather than in steps.

Program Change

The Program Change message is used to specify the type of instrument which should be used to play sounds on a given Channel. This message needs only one data byte which specifies the new program number.

Control Change

MIDI Control Change messages are used to control a wide variety of functions in a synthesizer. Control Change messages, like other MIDI Channel messages, should only affect the Channel number indicated in the status byte. The Control Change status byte is followed by one data byte indicating the "controller number", and a second byte which specifies the "control value". The controller number identifies which function of the synthesizer is to be controlled by the message. A complete list of assigned controllers is found in the MIDI 1.0 Detailed Specification.

- Bank Select

Controller number zero (with 32 as the LSB) is defined as the bank select. The bank select function is used in some synthesizers in conjunction with the MIDI Program Change message to expand the number of different instrument sounds which may be specified (the Program Change message alone allows selection of one of 128 possible program numbers). The additional sounds are selected by preceding the

Program Change message with a Control Change message which specifies a new value for Controller zero and Controller 32, allowing 16,384 banks of 128 sound each.

Since the MIDI specification does not describe the manner in which a synthesizer's banks are to be mapped to Bank Select messages, there is no standard way for a Bank Select message to select a specific synthesizer bank. Some manufacturers, such as Roland (with "GS") and Yamaha (with "XG") , have adopted their own practices to assure some standardization within their own product lines.

- RPN / NRPN

Controller number 6 (Data Entry), in conjunction with Controller numbers 96 (Data Increment), 97 (Data Decrement), 98 (Registered Parameter Number LSB), 99 (Registered Parameter Number MSB), 100 (Non-Registered Parameter Number LSB), and 101 (Non-Registered Parameter Number MSB), extend the number of controllers available via MIDI. Parameter data is transferred by first selecting the parameter number to be edited using controllers 98 and 99 or 100 and 101, and then adjusting the data value for that parameter using controller number 6, 96, or 97.

RPN and NRPN are typically used to send parameter data to a synthesizer in order to edit sound patches or other data. Registered parameters are those which have been assigned some particular function by the MIDI Manufacturers Association (MMA) and the Japan MIDI Standards Committee (JMISC). For example, there are Registered Parameter numbers assigned to control pitch bend sensitivity and master tuning for a synthesizer. Non-Registered parameters have not been assigned specific functions, and may be used for different functions by different manufacturers. Here again, Roland and Yamaha, among others, have adopted their own practices to assure some standardization.

Channel Mode Messages

Channel Mode messages (MIDI controller numbers 121 through 127) affect the way a synthesizer responds to MIDI data. Controller number 121 is used to reset all controllers. Controller number 122 is used to enable or disable Local Control (In a MIDI synthesizer which has it's own keyboard, the functions of the keyboard controller and the synthesizer can be isolated by turning Local Control off). Controller numbers 124 through 127 are used to select between Omni Mode On or Off, and to select between the Mono Mode or Poly Mode of operation.

When Omni mode is On, the synthesizer will respond to incoming MIDI data on all channels. When Omni mode is Off, the synthesizer will only respond to MIDI messages on one Channel. When Poly mode is selected, incoming Note On messages are played polyphonically. This means that when multiple Note On messages are received, each note is assigned its own voice (subject to the number of voices available in the synthesizer). The result is that multiple notes are played at the same time. When Mono mode is selected, a single voice is assigned per MIDI Channel. This means that only one note can be played on a given Channel at a given time. Most modern MIDI synthesizers will default to Omni On/Poly mode of operation. In this mode, the synthesizer will play note messages received on any MIDI Channel, and notes received on each Channel are played polyphonically. In the Omni Off/Poly mode of operation, the synthesizer will receive on a single Channel and play the notes received on this Channel polyphonically. This mode could be useful when several synthesizers are daisy-chained using MIDI THRU. In this case each synthesizer in the chain can be set to play one part (the MIDI data on one Channel), and ignore the information related to the other parts.

Note that a MIDI instrument has one MIDI Channel which is designated as its "Basic Channel". The Basic Channel assignment may be hard-wired, or it may be selectable. Mode messages can only be received by an instrument on the Basic Channel.

System Messages

MIDI System Messages are classified as being System Common Messages, System Real Time Messages, or System Exclusive Messages. System Common messages are intended for all receivers in the system. System Real Time messages are used for synchronization between clock-based MIDI components. System Exclusive messages include a Manufacturer's Identification (ID) code, and are used to transfer any number of data bytes in a format specified by the referenced manufacturer.

System Common Messages

The System Common Messages which are currently defined include MTC Quarter Frame, Song Select, Song Position Pointer, Tune Request, and End Of Exclusive (EOX). The MTC Quarter Frame message is part of the MIDI Time Code information used for synchronization of MIDI equipment and other equipment, such as audio or video tape machines.

The Song Select message is used with MIDI equipment, such as sequencers or drum machines, which can store and recall a number of different songs. The Song Position Pointer is used to set a sequencer to start playback of a song at some point other than at the beginning. The Song Position Pointer value is related to the number of MIDI clocks which would have elapsed between the beginning of the song and the desired point in the song. This message can only be used with equipment which recognizes MIDI System Real Time Messages (MIDI Sync).

The Tune Request message is generally used to request an analog synthesizer to retune its' internal oscillators. This message is generally not needed with digital synthesizers.

The EOX message is used to flag the end of a System Exclusive message, which can include a variable number of data bytes.

System Real Time Messages

The MIDI System Real Time messages are used to synchronize all of the MIDI clock-based equipment within a system, such as sequencers and drum machines. Most of the System Real Time messages are normally ignored by keyboard instruments and synthesizers. To help ensure accurate timing, System Real Time messages are given priority over other messages, and these single-byte messages may occur anywhere in the data stream (a Real Time message may appear between the status byte and data byte of some other MIDI message).

The System Real Time messages are the Timing Clock, Start, Continue, Stop, Active Sensing, and the System Reset message. The Timing Clock message is the master clock which sets the tempo for playback of a sequence. The Timing Clock message is sent 24 times per quarter note. The Start, Continue, and Stop messages are used to control playback of the sequence.

The Active Sensing signal is used to help eliminate "stuck notes" which may occur if a MIDI cable is disconnected during playback of a MIDI sequence. Without Active Sensing, if a cable is disconnected

during playback, then some notes may be left playing indefinitely because they have been activated by a Note On message, but the corresponding Note Off message will never be received.

The System Reset message, as the name implies, is used to reset and initialize any equipment which receives the message. This message is generally not sent automatically by transmitting devices, and must be initiated manually by a user.

System Exclusive Messages

System Exclusive messages may be used to send data such as patch parameters or sample data between MIDI devices. Manufacturers of MIDI equipment may define their own formats for System Exclusive data. Manufacturers are granted unique identification (ID) numbers by the MMA or the JMSC, and the manufacturer ID number is included as part of the System Exclusive message. The manufacturer ID is followed by any number of data bytes, and the data transmission is terminated with the EOX message. Manufacturers are required to publish the details of their System Exclusive data formats, and other manufacturers may freely utilize these formats, provided that they do not alter or utilize the format in a way which conflicts with the original manufacturers specifications.

Certain System Exclusive ID numbers are reserved for special protocols. Among these are the MIDI Sample Dump Standard, which is a System Exclusive data format defined in the MIDI specification for the transmission of sample data between MIDI devices, as well as MIDI Show Control and MIDI Machine Control.

Running Status

Since MIDI data is transmitted serially, it is possible that musical events which originally occurred at the same time and must be sent one at a time in the MIDI data stream may not actually be played at exactly the same time. With a data transmission rate of 31.25 Kbit/s and 10 bits transmitted per byte of MIDI data, a 3-byte Note On or Note Off message takes about 1 ms to be sent, which is generally short enough that the events are perceived as having occurred simultaneously. In fact, for a person playing a MIDI instrument keyboard, the time skew between playback of notes when 10 keys are pressed simultaneously should not exceed 10 ms, and this would not be perceptible.

However, MIDI data being sent from a sequencer can include a number of different parts. On a given beat, there may be a large number of musical events which should occur simultaneously, and the delays introduced by serialization of this information might be noticeable. To help reduce the amount of data transmitted in the MIDI data stream, a technique called "running status" may be employed.

Running status considers the fact that it is very common for a string of consecutive messages to be of the same message type. For instance, when a chord is played on a keyboard, 10 successive Note On messages may be generated, followed by 10 Note Off messages. When running status is used, a status byte is sent for a message only when the message is not of the same type as the last message sent on the same Channel. The status byte for subsequent messages of the same type may be omitted (only the data bytes are sent for these subsequent messages).

The effectiveness of running status can be enhanced by sending Note On messages with a velocity of zero in place of Note Off messages. In this case, long strings of Note On messages will often occur. Changes in some of the MIDI controllers or movement of the pitch bend wheel on a musical instrument

can produce a staggering number of MIDI Channel voice messages, and running status can also help a great deal in these instances.

MIDI Sequencers and Standard MIDI Files

MIDI messages are received and processed by a MIDI synthesizer in real time. When the synthesizer receives a MIDI "note on" message it plays the appropriate sound. When the corresponding "note off" message is received, the synthesizer turns the note off. If the source of the MIDI data is a musical instrument keyboard, then this data is being generated in real time. When a key is pressed on the keyboard, a "note on" message is generated in real time. In these real time applications, there is no need for timing information to be sent along with the MIDI messages.

However, if the MIDI data is to be stored as a data file, and/or edited using a sequencer, then some form of "time-stamping" for the MIDI messages is required. The Standard MIDI Files specification provides a standardized method for handling time-stamped MIDI data. This standardized file format for time-stamped MIDI data allows different applications, such as sequencers, scoring packages, and multimedia presentation software, to share MIDI data files.

The specification for Standard MIDI Files defines three formats for MIDI files. MIDI sequencers can generally manage multiple MIDI data streams, or "tracks". Standard MIDI files using Format 0 store all of the MIDI sequence data in a single track. Format 1 files store MIDI data as a collection of tracks. Format 2 files can store several independent patterns. Format 2 is generally not used by MIDI sequencers for musical applications. Most sophisticated MIDI sequencers can read either Format 0 or Format 1 Standard MIDI Files. Format 0 files may be smaller, and thus conserve storage space. They may also be transferred using slightly less system bandwidth than Format 1 files. However, Format 1 files may be viewed and edited more directly, and are therefore generally preferred.

Synthesizer Basics

Polyphony

The polyphony of a sound generator refers to its ability to play more than one note at a time. Polyphony is generally measured or specified as a number of notes or voices. Most of the early music synthesizers were monophonic, meaning that they could only play one note at a time. If you pressed five keys simultaneously on the keyboard of a monophonic synthesizer, you would only hear one note. Pressing five keys on the keyboard of a synthesizer which was polyphonic with four voices of polyphony would, in general, produce four notes. If the keyboard had more voices (many modern sound modules have 16, 24, or 32 note polyphony), then you would hear all five of the notes.

Sounds

The different sounds that a synthesizer or sound generator can produce are sometimes called "patches", "programs", "algorithms", or "timbres". Programmable synthesizers commonly assign "program

numbers" (or patch numbers) to each sound. For instance, a sound module might use patch number 1 for its acoustic piano sound, and patch number 36 for its fretless bass sound. The association of all patch numbers to all sounds is often referred to as a patch map.

Via MIDI, a Program Change message is used to tell a device receiving on a given Channel to change the instrument sound being used. For example, a sequencer could set up devices on Channel 4 to play fretless bass sounds by sending a Program Change message for Channel four with a data byte value of 36 (this is the General MIDI program number for the fretless bass patch).

Multitimbral Mode

A synthesizer or sound generator is said to be multitimbral if it is capable of producing two or more different instrument sounds simultaneously. If a synthesizer can play five notes simultaneously, and it can produce a piano sound and an acoustic bass sound at the same time, then it is multitimbral. With enough notes of polyphony and "parts" (multitimbral) a single synthesizer could produce the entire sound of a band or orchestra.

Multitimbral operation will generally require the use of a sequencer to send the various MIDI messages required. For example, a sequencer could send MIDI messages for a piano part on Channel 1, bass on Channel 2, saxophone on Channel 3, drums on Channel 10, etc. A 16 part multitimbral synthesizer could receive a different part on each of MIDI's 16 logical channels.

The polyphony of a multitimbral synthesizer is usually allocated dynamically among the different parts (timbres) being used. At any given instant five voices might be needed for the piano part, two voices for the bass, one for the saxophone, plus 6 voices for the drums. Note that some sounds on some synthesizers actually utilize more than one "voice", so the number of notes which may be produced simultaneously may be less than the stated polyphony of the synthesizer, depending on which sounds are being utilized.

The General MIDI (GM) System

At the beginning of a MIDI sequence, a Program Change message is usually sent on each Channel used in the piece in order to set up the appropriate instrument sound for each part. The Program Change message tells the synthesizer which patch number should be used for a particular MIDI Channel. If the synthesizer receiving the MIDI sequence uses the same patch map (the assignment of patch numbers to sounds) that was used in the composition of the sequence, then the sounds will be assigned as intended.

Prior to General MIDI, there was no standard for the relationship of patch numbers to specific sounds for synthesizers. Thus, a MIDI sequence might produce different sounds when played on different synthesizers, even though the synthesizers had comparable types of sounds. For example, if the composer had selected patch number 5 for Channel 1, intending this to be an electric piano sound, but the synthesizer playing the MIDI data had a tuba sound mapped at patch number 5, then the notes intended for the piano would be played on the tuba when using this synthesizer (even though this synthesizer may have a fine electric piano sound available at some other patch number).

The General MIDI (GM) Specification defines a set of general capabilities for General MIDI

Instruments. The General MIDI Specification includes the definition of a General MIDI Sound Set (a patch map), a General MIDI Percussion map (mapping of percussion sounds to note numbers), and a set of General MIDI Performance capabilities (number of voices, types of MIDI messages recognized, etc.). A MIDI sequence which has been generated for use on a General MIDI Instrument should play correctly on any General MIDI synthesizer or sound module.

The General MIDI system utilizes MIDI Channels 1-9 and 11-16 for chromatic instrument sounds, while Channel number 10 is utilized for "key-based" percussion sounds. These instrument sounds are grouped into "sets" of related sounds. For example, program numbers 1-8 are piano sounds, 9-16 are chromatic percussion sounds, 17-24 are organ sounds, 25-32 are guitar sounds, etc.

For the instrument sounds on channels 1-9 and 11-16, the note number in a Note On message is used to select the pitch of the sound which will be played. For example if the Vibraphone instrument (program number 12) has been selected on Channel 3, then playing note number 60 on Channel 3 would play the middle C note (this would be the default note to pitch assignment on most instruments), and note number 59 on Channel 3 would play B below middle C. Both notes would be played using the Vibraphone sound.

The General MIDI percussion sounds are set on Channel 10. For these "key-based" sounds, the note number data in a Note On message is used differently. Note numbers on Channel 10 are used to select which drum sound will be played. For example, a Note On message on Channel 10 with note number 60 will play a Hi Bongo drum sound. Note number 59 on Channel 10 will play the Ride Cymbal 2 sound.

It should be noted that the General MIDI system specifies sounds using program numbers 1 through 128. The MIDI Program Change message used to select these sounds uses an 8-bit byte, which corresponds to decimal numbering from 0 through 127, to specify the desired program number. Thus, to select GM sound number 10, the Glockenspiel, the Program Change message will have a data byte with the decimal value 9.

The General MIDI system specifies which instrument or sound corresponds with each program/patch number, but General MIDI does not specify how these sounds are produced. Thus, program number 1 should select the Acoustic Grand Piano sound on any General MIDI instrument. However, the Acoustic Grand Piano sound on two General MIDI synthesizers which use different synthesis techniques may sound quite different.

Synthesis Technology: FM and Wavetable

There are a number of different technologies or algorithms used to create sounds in music synthesizers. Two widely used techniques are Frequency Modulation (FM) synthesis and Wavetable synthesis.

FM synthesis techniques generally use one periodic signal (the modulator) to modulate the frequency of another signal (the carrier). If the modulating signal is in the audible range, then the result will be a significant change in the timbre of the carrier signal. Each FM voice requires a minimum of two signal generators. These generators are commonly referred to as "operators", and different FM synthesis implementations have varying degrees of control over the operator parameters.

Sophisticated FM systems may use 4 or 6 operators per voice, and the operators may have adjustable envelopes which allow adjustment of the attack and decay rates of the signal. Although FM systems were implemented in the analog domain on early synthesizer keyboards, modern FM synthesis implementations are done digitally.

FM synthesis techniques are very useful for creating expressive new synthesized sounds. However, if the goal of the synthesis system is to recreate the sound of some existing instrument, this can generally be done more accurately with digital sample-based techniques.

Digital sampling systems store high quality sound samples digitally, and then replay these sounds on demand. Digital sample-based synthesis systems may employ a variety of special techniques, such as sample looping, pitch shifting, mathematical interpolation, and digital filtering, in order to reduce the amount of memory required to store the sound samples (or to get more types of sounds from a given amount of memory). These sample-based synthesis systems are often called "wavetable" synthesizers (the sample memory in these systems contains a large number of sampled sound segments, and can be thought of as a "table" of sound waveforms which may be looked up and utilized when needed).

Wavetable Synthesis Techniques

The majority of professional synthesizers available today use some form of sampled-sound or Wavetable synthesis. The trend for multimedia sound products is also towards wavetable synthesis. To help prospective MIDI developers, a number of the techniques employed in this type of synthesis are discussed in the following paragraphs.

Looping and Envelope Generation

One of the primary techniques used in wavetable synthesizers to conserve sample memory space is the looping of sampled sound segments. For many instrument sounds, the sound can be modeled as consisting of two major sections: the attack section and the sustain section. The attack section is the initial part of the sound, where the amplitude and the spectral characteristics of the sound may be changing very rapidly. The sustain section of the sound is that part of the sound following the attack, where the characteristics of the sound are changing less dynamically.

Figure 4 shows a waveform with portions which could be considered the attack and the sustain sections indicated. In this example, the spectral characteristics of the waveform remain constant throughout the sustain section, while the amplitude is decreasing at a fairly constant rate. This is an exaggerated example, in most natural instrument sounds, both the spectral characteristics and the amplitude continue to change through the duration of the sound. The sustain section, if one can be identified, is that section for which the characteristics of the sound are relatively constant.

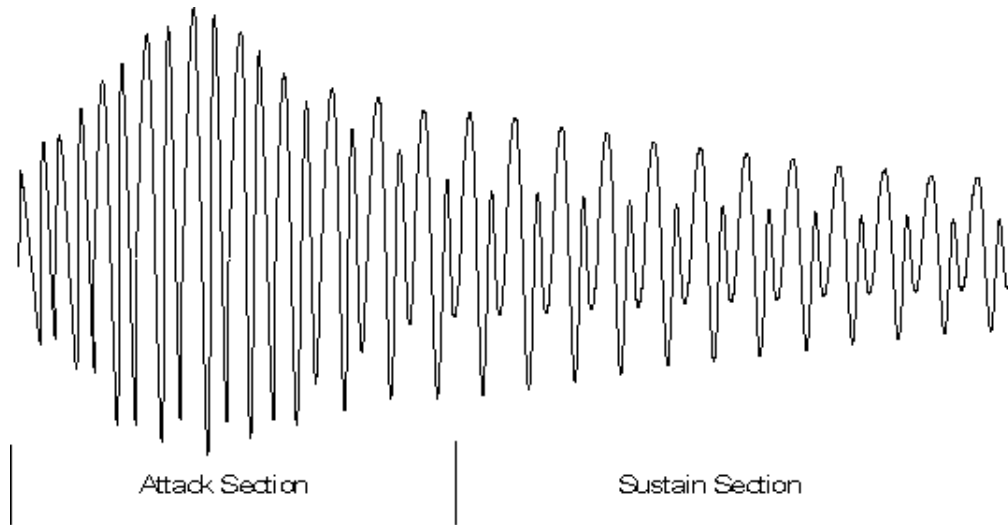


Figure 4: Attack and Sustain Portions of a Waveform

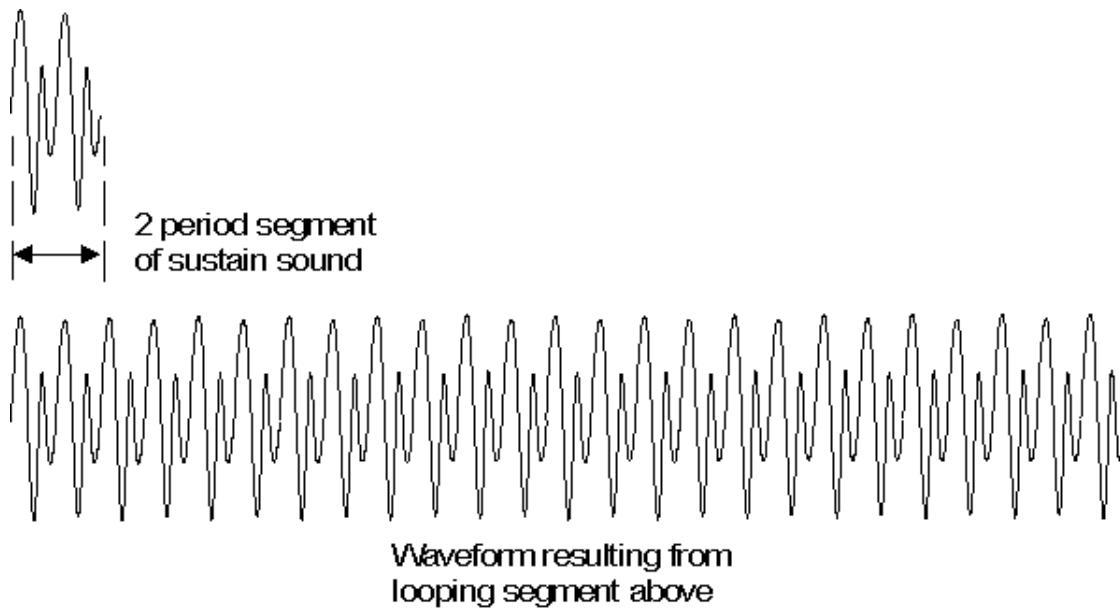


Figure 5: Looping of a Sample Segment

A great deal of memory can be saved in wavetable synthesis systems by storing only a short segment of the sustain section of the waveform, and then looping this segment during playback. Figure 5 shows a two period segment of the sustain section from the waveform in Figure 4, which has been looped to create a steady state signal. If the original sound had a fairly constant spectral content and amplitude during the sustained section, then the sound resulting from this looping operation should be a good approximation of the sustained section of the original.

For many acoustic string instruments, the spectral characteristics of the sound remain fairly constant during the sustain section, while the amplitude of the signal decays. This can be simulated with a looped segment by multiplying the looped samples by a decreasing gain factor during playback to get the desired shape or envelope. The amplitude envelope of a sound is commonly modeled as consisting of

some number of linear segments. An example is the commonly used four part piecewise-linear Attack-Decay-Sustain-Release (ADSR) envelope model. Figure 6 depicts a typical ADSR envelope shape, and Figure 7 shows the result of applying this envelope to the looped waveform from Figure 5.

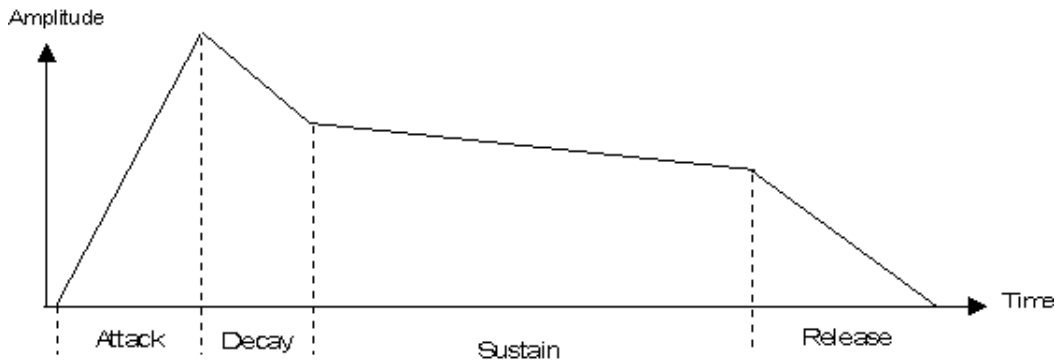


Figure 6: A Typical ADSR Amplitude Envelope

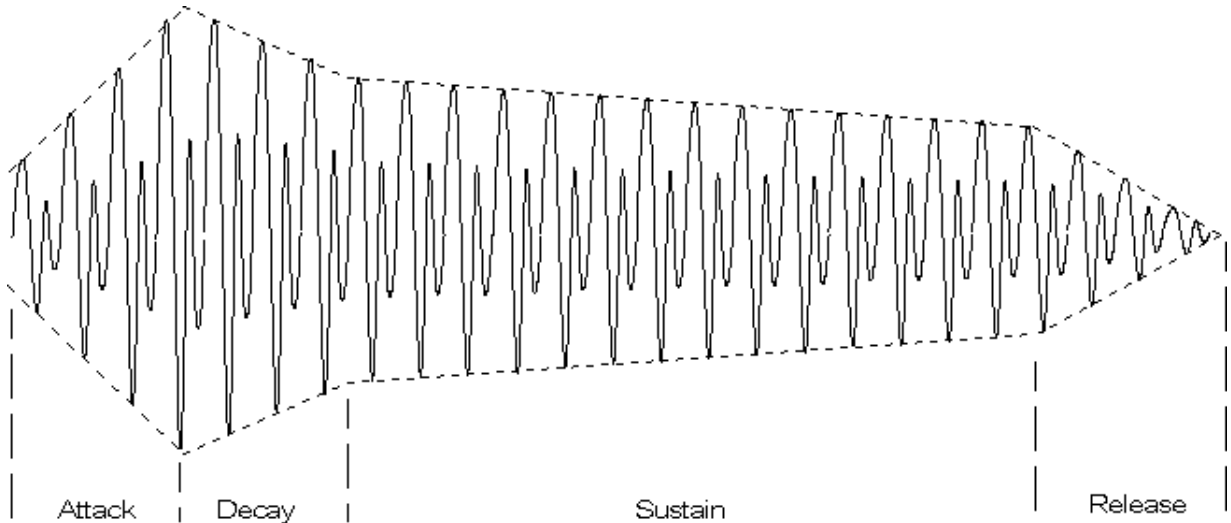


Figure 7: ADSR Envelope Applied to a Looped Sample Segment

A typical wavetable synthesis system would store sample data for the attack section and the looped section of an instrument sound. These sample segments might be referred to as the initial sound and the loop sound. The initial sound is played once through, and then the loop sound is played repetitively until the note ends. An envelope generator function is used to create an envelope which is appropriate for the particular instrument, and this envelope is applied to the output samples during playback.

Playback of the initial wave (with the attack portion of the envelope applied) begins when a Note On message is received. The length of the initial sound segment is fixed by the number of samples in the segment, and the length of the attack and decay sections of the envelope are generally also fixed for a given instrument sound.

The sustain section will continue to repeat the loop samples while applying the sustain envelope slope (which decays slowly in our examples), until a Note Off message is applied. The Note Off message

triggers the beginning of the release portion of the envelope.

Loop Length

The loop length is measured as a number of samples, and the length of the loop should be equal to an integral number of periods of the fundamental pitch of the sound being played (if this is not true, then an undesirable "pitch shift" will occur during playback when the looping begins). In practice, the length of the loop segment for an acoustic instrument sample may be many periods with respect to the fundamental pitch of the sound. If the sound has a natural vibrato or chorus effect, then it is generally desirable to have the loop segment length be an integral multiple of the period of the vibrato or chorus.

One-Shot Sounds

The previous paragraphs discussed dividing a sampled sound into an attack section and a sustain section, and then using looping techniques to minimize the storage requirements for the sustain portion. However, some sounds, particularly sounds of short duration or sounds whose characteristics change dynamically throughout their duration, are not suitable for looped playback techniques. Short drum sounds often fit this description. These sounds are stored as a single sample segment which is played once through with no looping. This class of sounds are referred to as "one-shot" sounds.

Sample Editing and Processing

There are a number of sample editing and processing steps involved in preparing sampled sounds for use in a wavetable synthesis system. The requirements for editing the original sample data to identify and extract the initial and loop segments have already been mentioned.

Editing may also be required to make the endpoints of the loop segment compatible. If the amplitude and the slope of the waveform at the beginning of the loop segment do not match those at the end of the loop, then a repetitive "glitch" will be heard during playback of the looped section. Additional processing may be performed to "compress" the dynamic range of the sound to improve the signal/quantizing noise ratio or to conserve sample memory. This topic is addressed next.

When all of the sample processing has been completed, the resulting sampled sound segments for the various instruments are tabulated to form the sample memory for the synthesizer.

Sample Data Compression

The signal-to-quantizing noise ratio for a digitally sampled signal is limited by sample word size (the number of bits per sample), and by the amplitude of the digitized signal. Most acoustic instrument sounds reach their peak amplitude very quickly, and the amplitude then slowly decays from this peak. The ear's sensitivity dynamically adjusts to signal level. Even in systems utilizing a relatively small sample word size, the quantizing noise level is generally not perceptible when the signal is near maximum amplitude. However, as the signal level decays, the ear becomes more sensitive, and the noise level will appear to increase. Of course, using a larger word size will reduce the quantizing noise, but there is a considerable price penalty paid if the number of samples is large.

Compression techniques may be used to improve the signal-to-quantizing noise ratio for some sampled sounds. These techniques reduce the dynamic range of the sound samples stored in the sample memory.

The sample data is decompressed during playback to restore the dynamic range of the signal. This allows the use of sample memory with a smaller word size (smaller dynamic range) than is utilized in the rest of the system. There are a number of different compression techniques which may be used to compress the dynamic range of a signal.

Note that there is some compression effect inherent in the looping techniques described earlier. If the loop segment is stored at an amplitude level which makes full use of the dynamic range available in the sample memory, and the processor and D/A converters used for playback have a wider dynamic range than the sample memory, then the application of a decay envelope during playback will have a decompression effect similar to that described in the previous paragraph.

Pitch Shifting

In order to minimize sample memory requirements, wavetable synthesis systems utilize pitch shifting, or pitch transposition techniques, to generate a number of different notes from a single sound sample of a given instrument. For example, if the sample memory contains a sample of a middle C note on the acoustic piano, then this same sample data could be used to generate the C# note or D note above middle C using pitch shifting.

Pitch shifting is accomplished by accessing the stored sample data at different rates during playback. For example, if a pointer is used to address the sample memory for a sound, and the pointer is incremented by one after each access, then the samples for this sound would be accessed sequentially, resulting in some particular pitch. If the pointer increment was two rather than one, then only every second sample would be played, and the resulting pitch would be shifted up by one octave (the frequency would be doubled).

In the previous example, the sample memory address pointer was incremented by an integer number of samples. This allows only a limited set of pitch shifts. In a more general case, the memory pointer would consist of an integer part and a fractional part, and the increment value could be a fractional number of samples. The memory pointer is often referred to as a "phase accumulator" and the increment value is then the "phase increment". The integer part of the phase accumulator is used to address the sample memory, the fractional part is used to maintain frequency accuracy.

For example if the phase increment value was equivalent to $1/2$, then the pitch would be shifted down by one octave (the frequency would be halved). A phase increment value of 1.05946 (the twelfth root of two) would create a pitch shift of one musical half-step (i.e. from C to C#) compared with an increment of 1. When non-integer increment values are utilized, the frequency resolution for playback is determined by the number of bits used to represent the fractional part of the address pointer and the address increment parameter.

Interpolation

When the fractional part of the address pointer is non-zero, then the "desired value" falls between available data samples. Figure 8 depicts a simplified addressing scheme wherein the Address Pointer and the increment parameter each have a 4-bit integer part and a 4-bit fractional part. In this case, the increment value is equal to $1 \frac{1}{2}$ samples. Very simple systems might simply ignore the fractional part of the address when determining the sample value to be sent to the D/A converter. The data values sent to the D/A converter when using this approach are indicated in the Figure 8, case I.

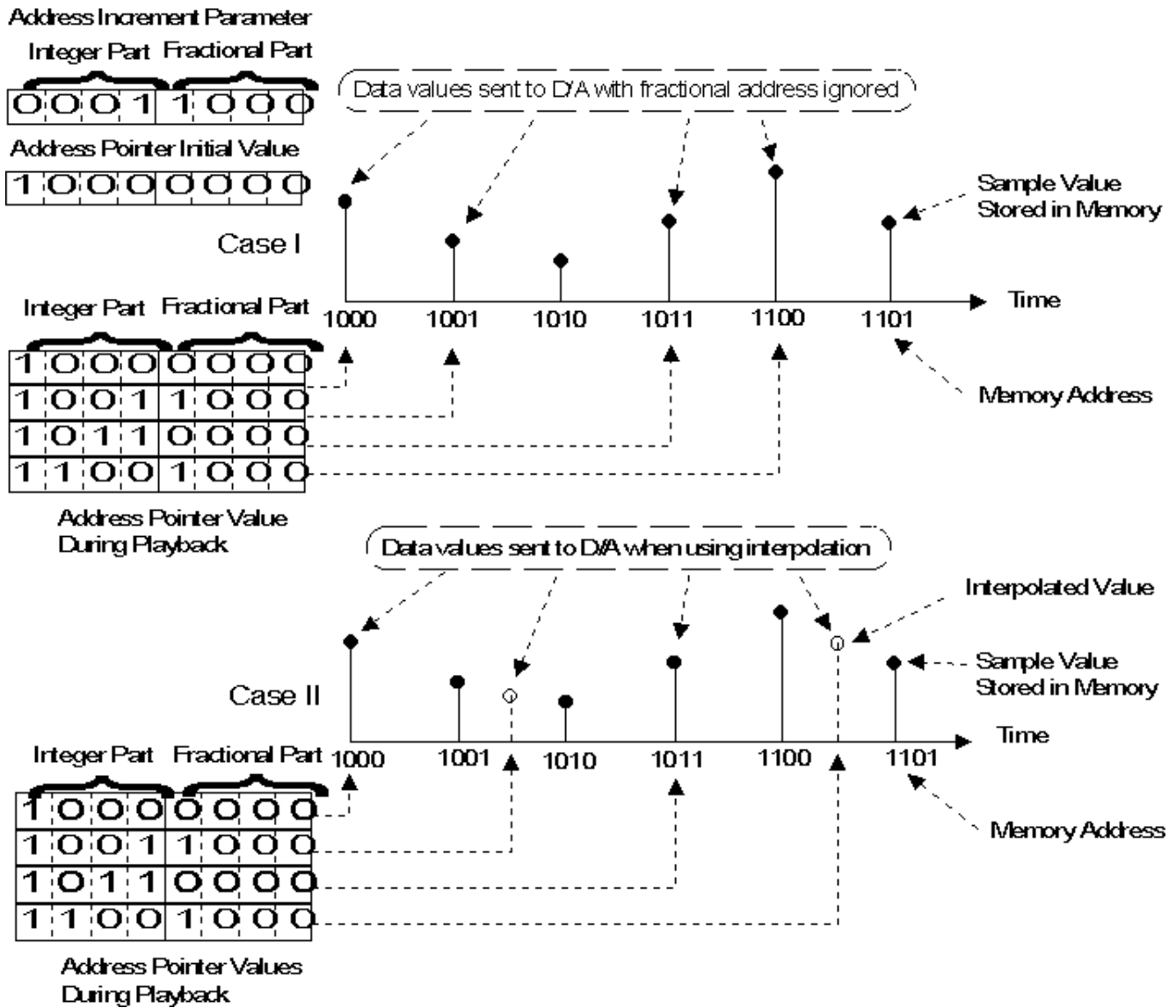


Figure 8: Sample Memory Addressing and Interpolation

A slightly better approach would be to use the nearest available sample value. More sophisticated systems would perform some type of mathematical interpolation between available data points in order to get a value to be used for playback. Values which might be sent to the D/A when interpolation is employed are shown as case II. Note that the overall frequency accuracy would be the same for both cases indicated, but the output is severely distorted in the case where interpolation is not used.

There are a number of different algorithms used for interpolation between sample values. The simplest is linear interpolation. With linear interpolation, interpolated value is simply the weighted average of the two nearest samples, with the fractional address used as a weighting constant. For example, if the address pointer indicated an address of $(n+K)$, where n is the integer part of the address and K is the fractional part, then the interpolated value can be calculated as $s(n+K) = (1-K)s(n) + (K)s(n+1)$, where $s(n)$ is the sample data value at address n . More sophisticated interpolation techniques can be utilized to further reduce distortion, but these techniques are computationally expensive.

Oversampling

Oversampling of the sound samples may also be used to improve distortion in wavetable synthesis systems. For example, if 4X oversampling were utilized for a particular instrument sound sample, then an address increment value of 4 would be used for playback with no pitch shift. The data points chosen during playback will be closer to the "desired values", on the average, than they would be if no oversampling were utilized because of the increased number of data points used to represent the waveform. Of course, oversampling has a high cost in terms of sample memory requirements.

In many cases, the best approach may be to utilize linear interpolation combined with varying degrees of oversampling where needed. The linear interpolation technique provides reasonable accuracy for many sounds, without the high penalty in terms of processing power required for more sophisticated interpolation methods. For those sounds which need better accuracy, oversampling is employed. With this approach, the additional memory required for oversampling is only utilized where it is most needed. The combined effect of linear interpolation and selective oversampling can produce excellent results.

Splits

When the pitch of a sampled sound is changed during playback, the timbre of the sound is changed somewhat also. The effect is less noticeable for small changes in pitch (up to a few semitones), than it is for a large pitch shift. To retain a natural sound, a particular sample of an instrument sound will only be useful for recreating a limited range of notes. To get coverage of the entire instrument range, a number of different samples, each with a limited range of notes, are used. The resulting instrument implementation is often referred to as a "multisampled" instrument. This technique can be thought of as splitting a musical instrument keyboard into a number of ranges of notes, with a different sound sample used for each range. Each of these ranges is referred to as a split, or key split.

Velocity splits refer to the use of different samples for different note velocities. Using velocity splits, one sample might be utilized if a particular note is played softly, where a different sample would be utilized for the same note of the same instrument when played with a higher velocity. This technique is not commonly used to produce basic sound samples because of the added memory expense, but both key splitting and velocity splitting techniques can be utilized as a performance enhancement. For instance, a key split might allow a fretless bass sound on the lower octaves of a keyboard, while the upper octaves play a vibraphone. Similarly, a velocity split might "layer" strings on top of an acoustic piano sound when the keys are hit with higher velocity.

Aliasing Noise

Earlier paragraphs discussed the timbre changes which result from pitch shifting. The resampling techniques used to shift the pitch of a stored sound sample can also result in the introduction of aliasing noise into an instrument sound. The generation of aliasing noise can also limit the amount of pitch shifting which may be effectively applied to a sound sample. Sounds which are rich in upper harmonic content will generally have more of a problem with aliasing noise. Low-pass filtering applied after interpolation can help eliminate the undesirable effect of aliasing noise. The use of oversampling also helps eliminate aliasing noise.

LFOs for Vibrato and Tremolo

Vibrato and tremolo are effects which are often produced by musicians playing acoustic instruments. Vibrato is basically a low-frequency modulation of the pitch of a note, while tremolo is modulation of the amplitude of the sound. These effects are simulated in synthesizers by implementing low-frequency oscillators (LFOs) which are used to modulate the pitch or amplitude of the synthesized sound being produced.

Natural vibrato and tremolo effects tend to increase in strength as a note is sustained. This is accomplished in synthesizers by applying an envelope generator to the LFO. For example, a flute sound might have a tremolo effect which begins at some point after the note has sounded, and the tremolo effect gradually increases to some maximum level, where it remains until the note stops sounding.

Layering

Layering refers to a technique in which multiple sounds are utilized for each note played. This technique can be used to generate very rich sounds, and may also be useful for increasing the number of instrument patches which can be created from a limited sample set. Note that layered sounds generally utilize more than one voice of polyphony for each note played, and thus the number of voices available is effectively reduced when these sounds are being used.

Digital Filtering

It was mentioned earlier that low-pass filtering may be used to help eliminate noise which may be generated during the pitch shifting process. There are also a number of ways in which digital filtering is used in the timbre generation process to improve the resulting instrument sound. In these applications, the digital filter implementation is polyphonic, meaning that a separate filter is implemented for each voice being generated, and the filter implementation should have dynamically adjustable cutoff frequency and/or Q.

For many acoustic instruments, the character of the tone which is produced changes dramatically as a function of the amplitude level at which the instrument is played. For example, the tone of an acoustic piano may be very bright when the instrument is played forcefully, but much more mellow when it is played softly. Velocity splits, which utilize different sample segments for different note velocities, can be implemented to simulate this phenomena.

Another very powerful technique is to implement a digital low-pass filter for each note with a cutoff frequency which varies as a function of the note velocity. This polyphonic digital filter dynamically adjusts the output frequency spectrum of the synthesized sound as a function of note velocity, allowing a very effective recreation of the acoustic instrument timbre.

Another important application of digital filtering is in smoothing out the transitions between samples in key-based splits. At the border between two splits, there will be two adjacent notes which are based on different samples. Normally, one of these samples will have been pitch shifted up to create the required note, while the other will have been shifted down in pitch. As a result, the timbre of these two adjacent notes may be significantly different, making the split obvious. This problem may be alleviated by employing a digital filter which uses the note number to control the filter characteristics. A table may be constructed containing the filter characteristics for each note number of a given instrument. The filter characteristics are chosen to compensate for the pitch shifting associated with the key splits used for that instrument.

It is also common to control the characteristics of the digital filter using an envelope generator or an LFO. The result is an instrument timbre which has a spectrum which changes as a function of time. An envelope generator might be used to control the filter cutoff frequency generate a timbre which is very bright at the onset, but which gradually becomes more mellow as the note decays. Sweeping the cutoff frequency of a filter with a high Q setting using an envelope generator or LFO can help when trying to simulate the sounds of analog synthesizers.

The PC to MIDI Connection

To use MIDI with a personal computer, a PC to MIDI interface product is generally required (there are a few personal computers which come equipped with built-in MIDI interfaces). There are a number of MIDI interface products for PCs. The most common types of MIDI interfaces for IBM compatibles are add-in cards which plug into an expansion slot on the PC bus, but there are also serial port MIDI interfaces (connects to a serial port on the PC) and parallel port MIDI interfaces (connects to the PC printer port). Most other popular personal computers will use a serial port connection.

The fundamental function of a MIDI interface for the PC is to convert parallel data bytes from the PC data bus into the serial MIDI data format and vice versa (a UART function). However, "smart" MIDI interfaces may provide a number of more sophisticated functions, such as generation of MIDI timing data, MIDI data buffering, MIDI message filtering, synchronization to external tape machines, and more.

The specific interface design used has some specific importance to the multimedia market, due to the need for essentially transparent operation of games and other applications which use General MIDI. GM does not define how the game is supposed to connect with the synthesizer, so sound-card standards are also needed to assure proper operation. While some PC operating systems provide device independence, this is not true of the typical IBM-PC running MS-DOS, where hardware MIDI interface standards are required.

The defacto standard for MIDI interface add-in cards for the IBM-PC is the Roland MPU-401 interface. The MPU-401 is a smart MIDI interface, which also supports a dumb mode of operation (often referred to as "UART mode"). There are a number of MPU-401 compatible MIDI interfaces on the market, some which only support the UART (dumb) mode of operation. In addition, many IBM-PC add-in sound cards include built-in MIDI interfaces which implement the UART mode functions of the MPU-401.

PC Compatibility Issues

There are two levels of compatibility which must be considered for MIDI applications running on the PC. First is the compatibility of the application with the MIDI interface being used. The second is the compatibility of the application with the MIDI synthesizer. For the purposes of this tutorial we will be talking only about IBM-PC and compatible systems, though much of this information can also be applied to other PC systems. Compatibility considerations under DOS and the Microsoft Windows operating system are discussed in the following paragraphs.

MS-DOS Applications

MS-DOS applications which utilize MIDI synthesizers include MIDI sequencing software, music scoring applications, and a variety of games. In terms of MIDI interface compatibility, virtually all of these applications support the MPU-401 interface, and most only require the UART mode. These applications should work correctly on any compatible PC equipped with a MPU-401, a full-featured MPU-401 compatible, or a sound card with a MPU-401 UART-mode capability. Other MIDI interfaces, such as serial port or parallel port MIDI adapters, will only work if the application provides support for that particular model of MIDI interface.

A particular application may provide support for a number of different models of synthesizers or sound modules. Prior to the General MIDI standard, there was no widely accepted standard patch set for synthesizers, so applications generally needed to provide support for each of the most popular synthesizers at the time. If the application did not support the particular model of synthesizer or sound module that was attached to the PC, then the sounds produced by the application might not be the sounds which were intended. Modern applications can provide support for a General MIDI (GM) synthesizer, and any GM-compatible sound source should produce the correct sounds.

Multimedia PC (MPC) Systems

The number of applications for high quality audio functions on the PC (including music synthesis) grew explosively after the introduction of Microsoft Windows 3.0 with Multimedia Extensions ("Windows with Multimedia") in 1991. These extensions are also incorporated into the Windows 3.1 operating system. The Multimedia PC (MPC) specification, originally published by Microsoft in 1991, is now published and maintained by the Multimedia PC Marketing Council, a subsidiary of the Software Publishers Association. The MPC specification states the minimum requirements for multimedia-capable Personal Computers to ensure compatibility in running multimedia applications based on Windows 3.1 or Windows with Multimedia.

The audio capabilities of an MPC system must include digital audio recording and playback (linear PCM sampling), music synthesis, and audio mixing. The current MPC specifications define two different levels of performance. The requirements for music synthesizers in MPC level 1 and MPC level 2 systems are essentially the same, although the digital audio recording and playback requirements for MPC level 1 and MPC level 2 compliance are different.

For MIDI, the current MPC specifications attempt to balance performance and cost issues by defining two types of synthesizers; a "Base Multitimbral Synthesizer", and an "Extended Multitimbral Synthesizer". Both the Base and the Extended synthesizer are expected to use a General MIDI patch set, but neither actually meets the full requirements of General MIDI polyphony or simultaneous timbres. Base Multitimbral Synthesizers must be capable of playing 6 "melodic notes" and "2 percussive" notes simultaneously, using 3 "melodic timbres" and 2 "percussive timbres".

The formal requirements for an Extended Multitimbral Synthesizer are only that it must have capabilities which exceed those specified for a Base Multitimbral Synthesizer. However, the "goals" for an Extended synthesizer include the ability to play 16 melodic notes and 8 percussive notes simultaneously, using 9 melodic timbres and 8 percussive timbres.

The MPC specification also includes an authoring standard for MIDI composition. This standard requires that each MIDI file contain two arrangements of the same song, one for Base synthesizers and one for Extended synthesizers, allowing for differences in available polyphony and timbres. The MIDI data for the Base synthesizer arrangement is sent on MIDI channels 13 - 16 (with the percussion track on Channel 16), and the Extended synthesizer arrangement utilizes channels 1 - 10 (percussion is on Channel 10).

This technique is intended to optimize the MIDI file to play on both types of synthesizer, but is also a potential source of problems for GM synthesizers. A GM synthesizer will receive on all 16 Channels and subsequently play both performances, including playing the Channel 16 percussion track, but with a melodic instrument.

Microsoft has addressed this issue for future versions of Windows by recommending the full General MIDI model instead of the Base/Extended model. However, existing MIDI data which has been authored for the Microsoft dual-format will continue to be a problem for next-generation Windows systems, and is a problem in any system today that contains a full GM-compatible synthesizer.

The only current solution is to use the Windows MIDI mapper, as described below, to block the playback of the extra Channels. Unfortunately, this will also result in blocking needed data on those same Channels in a GM-compatible score. The ideal solution might be to develop a scheme for identifying Standard MIDI Files containing base/extended data, and to provide a "dynamic" MIDI mapping scheme which takes into account the type of file being played. This approach could also be applied to other standardized formats which offer some small problems for GM hardware, such as Roland's GS and Yamaha's XG formats.

Microsoft Windows Configuration

Windows applications address hardware devices such as MIDI interfaces or synthesizers through the use of drivers. The drivers provide applications software with a common interface through which hardware may be accessed, and this simplifies the hardware compatibility issue. Synthesizer drivers must be installed using the Windows Driver applet within the Control Panel.

When a MIDI interface or synthesizer is installed in the PC and a suitable device driver has been loaded, the Windows MIDI Mapper applet will then appear within the Control Panel. MIDI messages are sent from an application to the MIDI Mapper, which then routes the messages to the appropriate device driver. The MIDI Mapper may be set to perform some filtering or translations of the MIDI messages in route from the application to the driver. The processing to be performed by the MIDI Mapper is defined in the MIDI Mapper Setups, Patch Maps, and Key Maps.

MIDI Mapper Setups are used to assign MIDI channels to device drivers. For instance, If you have an MPU-401 interface with a General MIDI synthesizer and you also have a Creative Labs Sound Blaster card in your system, you might wish to assign channels 13 to 16 to the Ad Lib driver (which will drive the Base-level FM synthesizer on the Sound Blaster), and assign channels 1 - 10 to the MPU-401 driver. In this case, MPC compatible MIDI files will play on both the General MIDI synthesizer and the FM synthesizer at the same time. The General MIDI synthesizer will play the Extended arrangement on MIDI channels 1 - 10, and the FM synthesizer will play the Base arrangement on channels 13-16.

The MIDI Mapper Setups can also be used to change the Channel number of MIDI messages. If you have MIDI files which were composed for a General MIDI instrument, and you are playing them on a Base Multitimbral Synthesizer, you would probably want to take the MIDI percussion data coming from your application on Channel 10 and send this information to the device driver on Channel 16.

The MIDI Mapper patch maps are used to translate patch numbers when playing MPC or General MIDI files on synthesizers which do not use the General MIDI patch numbers. Patch maps can also be used to play MIDI files which were arranged for non-GM synthesizers on GM synthesizers. For example, the Windows-supplied MT-32 patch map can be used when playing GM-compatible .MID files on the Roland MT-32 sound module or LAPC-1 sound card. The MIDI Mapper key maps perform a similar function, translating the key numbers contained in MIDI Note On and Note Off messages. This capability is useful for translating GM-compatible percussion parts for playback on non-GM synthesizers or vice-versa. The Windows-supplied MT-32 key map changes the key-to-drum sound assignments used for General MIDI to those used by the MT-32 and LAPC-1.

Summary

The MIDI protocol provides an efficient format for conveying musical performance data, and the Standard MIDI Files specification ensures that different applications can share time-stamped MIDI data. While this alone is largely sufficient for the working MIDI musician, the storage efficiency and on-the-fly editing capability of MIDI data also makes MIDI an attractive vehicle for generation of sounds in multimedia applications, computer games, or high-end karaoke equipment.

The General MIDI system provides a common set of capabilities and a common patch map for high polyphony, multitimbral synthesizers, providing musical sequence authors and multimedia applications developers with a common target platform for synthesis. With the greater realism which comes from wavetable synthesis, and as newer, interactive, applications come along, MIDI-driven synthesizers will continue to be an important component for sound generation devices and multimedia applications.

**Copyright 1995 MIDI Manufacturers Association. All rights reserved.
No part of this document may be reproduced or copied without written permission of the publisher.**

Elser

Word convertress

Very alpha Näherung of an independent Word 6.0 Document Format specification

Copyright © 1997 by Martin Schwartz

Note!

This document is some respects outdated!

I still don't know other sources for Word 6 and Word 7, but the format of Word 97 is published in the Knowledge Base of Microsoft.

Based on this *Caolan Mcnamara* developed MSWordView to convert Word 97 documents to HTML.

You also should be aware of the most important source for document formats: Wotsit.

Here *Rex Swain* explains how to get the binary format of Word 97:

The binary file format for Word 97 is available online. For more information, please see the MSDN Library Online at the following Microsoft Developer Network (MSDN) World Wide Web site:

<http://www.microsoft.com/msdn>

Once you subscribe to the MSDN, you can obtain an online copy of the Binary File Format. To do this, follow these steps:

1. On the MSDN World Wide Web site, click MSDN Library Online.
 2. Double-click Microsoft Office Development.
 3. Double-click Office.
 4. Double-click Microsoft Office 97 Binary File Formats.
 5. Double-click Microsoft Word 97 Binary File Format.
 6. Click to select the topic you want.
-

This document is part of Laola, a distribution dealing among other things with Word 6 style documents.

These are documents done with Word 6 or with Word for Windows '95 alias Word 7. This release of the specification is very first, and it is highly speculative. It is not valid for Word 8 documents. It's neither complete, nor systematically verified. It's rather buggy, strange, incomplete and it is probable that many things will be changed or at least renamed in future. Please be aware of this, when using.

If you are not familiar with Word, you might notice, that this description doesn't fit to the document you are inspecting. This is, because the "real" Word document is put into an envelope. You can retrieve the document from its envelope with Microsoft's OLE library, or by using Laola library. Laola HomePage can be found at:

<http://www.wbs.cs.tu-berlin.de/~schwartz/pmh/index.html>

or:

<http://user.cs.tu-berlin.de/~schwartz/pmh/index.html>

Last but not least a note about nomenclature: though I tried to check against freely available Microsoft documents like the RTF specification, I'm afraid in many cases I've simply been lacking of time, information or I misinterpreted information. I hope the degree of confusion will be sustainable.

Comments appreciated!

Martin Schwartz

Table of Contents

- 1. Data Types**
 - 2. Word Header**
 - 3. Anchors**
 - 4. Character Formats**
 - 5. Fastsave Info**
 - 6. Field Info**
 - 7. Font Info**
 - 8. Footnote Info**
 - 9. Format Codes**
 - 10. Macro Info**
 - 11. Paragraph Formats**
 - 12. Paragraph Info**
 - 13. PrinterEss Info**
 - 14. Style Sheet**
 - 15. Summary Information**
-

Data Types

type	Name	Structure		Information
		offset	type	
Arrange	thing(n)	00	thing [0..n-1]	An n element array of thing.
	thing()	00	thing [0..]	An array of thing, where the number of elements is not known.
	empty			Nothing. Takes 0 byte space.
	even	00	byte empty	0 on odd addresses. Empty on even addresses. This <i>even</i> makes an address even.
Offset	blocko	00	word	n fileo == n * 0x200
	fileo	00	long	Offset relative to begin of file
	texto	00	long	Offset relative to begin of text. Equals not (!) text_begin + texto
	footo	00	long	Offset relative to begin of footnote text. texto == text_len + footo
	desto	00	long	Offset relative to begin of destination text. texto == text_len + foot_len + desto
String	str(n)	00	char(n)	A n byte long text string.
	bstr	00 01	char str(n)	n A n byte long text string.
	wstr	00 02	word str(n)	n A n byte long text string.
	zstr	00 n	str(n) char 0	A n byte long text string, ending with a further zero byte.
Struct	pair	00 04	fileo long	Pointer to a structure. Size of a structure.
	struct	00	char()	An arbitrary sized array of bytes.

Word Header

offset	type	value	Information
00	long	word_id	Word identifier
04	long	word_rev	Word Revision identifier
08	blocko	next_doc	At this offset a new word header starts. Master doc feature?

0a	word	doc_status	Bitmask showing if document is a template, crypted and more
0c	word	0x65, 0x00	unknown
0e	word	pw_hash	Calculated from password. See program pwhash.
10	word	pw_key	Calculated from password. Unknown how. If one knows the crypt key, this value can be used to calculate the password. See routine get_password in word6::decrypt.pl
12	word	0	unknown
14	word	0	unknown
16	word	0	unknown
18	long	text_begin	Begin of first text chunk. Usually 0x300.
1c	long	text_end	End of first text chunk.
20	long	file_len	Size of document in bytes.
24	long	0	unknown
28	long	0	unknown
2c	long	0	unknown
30	long	0	unknown
34	long	text_len	Length of all text of this document. At not "fast saved" documents this often is text_end minus text_begin.
38	long	foot_len	Length of footnote text.
3c	long	dest_len	Length of "destination" text.
40	long	0	unknown
44	long		(something with pair 0x080)
48	long	0	unknown
4c	long		unknown
50	long	0	unknown

54	long	0	(something with pair 0x222)
58	pair	stylesheet	Style sheet.
60	pair	stylesheet_copy	Always same values as pair 0x58.
68	pair	footnote_info	Footnote info.
70	pair	footnote_offsets	Footnote offsets.
78	pair		unknown. comment_info?
80	pair		unknown, long()
88	pair	section_info	
90	pair	par_info	
98	pair		unknown
a0	pair		unknown
a8	pair		unknown
b0	pair	dest_offsets	
b8	pair	charf_block_info	Offset to character format block descriptions.
c0	pair	parf_block_info	Offset to paragraph format block descriptions.
c8	pair		unknown
d0	pair	font_info	
d8	pair	field_info	
e0	pair	dest_field_info	
e8	pair		unknown
f0	pair		unknown
f8	pair		unknown
100	pair	anchor_names	
108	pair	anchor_begin_o	
110	pair	anchor_end_o	
118	pair	macro_info	
120	pair		unknown
128	pair		unknown
130	pair	printer_info	
138	pair	printer 1	
140	pair	printer 2	

148	pair		unknown
150	pair	summary_info	
158	pair	summary_strings	
160	pair	fastsave_info	
168	pair		unknown
170	pair		unknown
178	pair		unknown. comment_info?
180	pair		unknown
188	word	0	
18a	blocko	charf_blocko_first	First char format offset
18c	blocko	parf_blocko_first	First par format offset
18e	word	charf_blocko_num	Number of char format offsets
190	word	parf_blocko_num	Number of par format offsets
192	pair		unknown
19a	pair		unknown
1a2	pair		unknown
1aa	pair		unknown
1b2	pair		unknown
1ba	pair		unknown
1c2	pair		unknown
1ca	pair		unknown
1d2	pair		unknown. math_info?
1da	pair		unknown. math_info?
1e2	pair		unknown
1ea	pair		unknown
1f2	pair		unknown
1fa	pair		former authresses
202	pair		unknown
20a	pair		unknown
212	pair		unknown
21a	pair		unknown
222	pair		unknown
22a	pair		unknown
232	pair		unknown

23a	pair		unknown
242	pair		unknown
242	pair		unknown
252	pair		unknown
25a	pair		unknown
262	pair		unknown
26a	pair		unknown
272	pair		unknown
27a	pair		unknown. math_info?
282	pair		unknown. math_info?
28a	pair		unknown
292	pair		unknown
29a	pair	history_info	Former authresses and paths.
2a2	pair		unknown

0x00 long	word_id
Value	Information
0065a5dc	
0068a5dc	
0068a697	
0068a699	
00c1a5ec	

0x04 long	word_rev
Value	Information
0407c02d	Word 6
0407c035	Word 6
0407c03d	Word 6.0c
0409c033	Word 6
0409c03d	Word 6
0409c03f	Word 6
0409c041	Word 6
0409c047	Word 6
0409e057	Word 7
0409e063	Word 7
04070049	Word 8

0x0a word		doc_status
Bit	Value	Document Information
00	TEMPLATE	Is a template
01		
02	FASTSAVE	Is stored in "fast save" format
03		Contains graphics?
04		
05		
06		
07		
08	CRYPT	Has a password
09		Unicode characters?
0a		
0b		
0c		
0d		
0e		
0f		

Style sheet

0x58 pair			stylesheet
Offset	Type	Value	Information
00	word	type	Type of style sheet. Following structure is valid for type 0x0e, only. (Have Unicode documents type 0x12?)
02	word		unknown
04	word	08	unknown
06	word	01	unknown
08	word	4b	unknown
0a	word	0f	unknown
0c	word	00	unknown
0e	word		unknown
10	struct()	stylesheet_entry	

stylesheet_entry			
Offset	Type	Value	Information
00	word	size	Size of entry -2. Value zero means, that this entry is not available. Rest of this entry then is missing.
02	byte	style_id	Style ID. 0xfe for styles defined by userEss.
03	byte	class	class & 0x0f == 0: standard style class & 0x0f == 0xf: userEss style class & 0xf0: unknown
04	word	prevstyle	bits 00..03: gender 1==paragraph, 2==character bits 04..11: number of prev style bits 12..15: state 0==valid, 0xf==invalid
06	word	nextstyle	bits 00..03: antigender 1==character, 2==paragraph bits 04..11: paragraphs: number of next style characters: this style's number bits 12..15: state always 0
08	word	size	Same as at offset 00.
0a	bstr	name	Name of style.
	even		
xx	struct	stylesheet par_entry	Paragraph format entry. See below. Only, if gender & 1.
	even		
yy	struct	stylesheet char_entry	Character format entry. See below. Always.

stylesheet_par_entry			
Offset	Type	Value	Information
00	word	size	Size of following data.
02	word	style_num	Style number.
04	str(size-2)	format	Paragraph format string.

stylesheet_char_entry			
Offset	Type	Value	Information
00	wstr	format	Character format string.

style_id			Information
Num	Type (english)	ID (deutsch)	
00	Normal	Standard	
01	Heading 1	Überschrift 1	
02	Heading 2	Überschrift 2	
03	Heading 3	Überschrift 3	
04	Heading 4	Überschrift 4	
05	Heading 5	Überschrift 5	
06	Heading 6	Überschrift 6	
07	Heading 7	Überschrift 7	
08	Heading 8	Überschrift 8	
09	Heading 9	Überschrift 9	
0a	Index 1	Index 1	
0b	Index 2	Index 2	
0c	Index 3	Index 3	
0d	Index 4	Index 4	
0e	Index 5	Index 5	
0f	Index 6	Index 6	
10	Index 7	Index 7	
11	Index 8	Index 8	
12	Index 9	Index 9	
13	TOC 1	Verzeichnis 1	
14	TOC 2	Verzeichnis 2	
15	TOC 3	Verzeichnis 3	
16	TOC 4	Verzeichnis 4	
17	TOC 5	Verzeichnis 5	
18	TOC 6	Verzeichnis 6	
19	TOC 7	Verzeichnis 7	
1a	TOC 8	Verzeichnis 8	
1b	TOC 9	Verzeichnis 9	
1c	Normal Indent	Standardeinzug	
1d	Footnote Text	Fußnotentext	
1e	Annotation Text	Anmerkungstext	
1f	Header	Kopfzeile	
20	Footer	Fußzeile	
21	Index Heading	Indexüberschrift	
22	Caption	Beschriftung	

23		Abbildungsverzeichnis	
24		Absenderadresse	
25		Umschlagadresse	
26	Footnote Reference	Fußnotenzeichen	
27	Annotation Reference	Anmerkungszeichen	
28	Line Number	Zeilennummer	
29	Page Number	Seitenzahl	
2a	Endnote Reference	Endnotenzeichen	
2b	Endnote Text	Endnotentext	
2c		Zusatz 2	
2d		Makrotext	
2e		Zusatz 1	
2f	List	Liste	
30	List Bullet,UL	Aufzählungszeichen,UL	
31	List Bullet,OL	Listennummer,OL	
32	List 2	Liste 2	
33	List 3	Liste 3	
34	List 4	Liste 4	
35	List 5	Liste 5	
36	List Bullet 2	Aufzählungszeichen 2	
37	List Bullet 3	Aufzählungszeichen 3	
38	List Bullet 4	Aufzählungszeichen 4	
39	List Bullet 5	Aufzählungszeichen 5	
3a	List Number 2	Listennummer 2	
3b	List Number 3	Listennummer 3	
3c	List Number 4	Listennummer 4	
3d	List Number 5	Listennummer 5	
3e	Title	Titel	
3f		Grußformel	
40		Unterschrift	
41	Default Paragraph Font	Absatz-Standardschriftart	
42	Body Text	Textkörper	
43		Textkörper-Einzug	
44		Listenfortsetzung	
45		Listenfortsetzung 2	
46		Listenfortsetzung 3	
47		Listenfortsetzung 4	

48		Listenfortsetzung 5	
49		Nachrichtenkopf	
4a	Subtitle	Untertitel	

Footnote Info

The footnote texts are stored regularly in the text section.

0x68 pair			footnote_info
Offset	Type	Value	Information
00	texto(n+1)	anchor	Offset to footnotes anchor. This is, where the footnote mark will be. $n = (size-4)/6$
xx	word(n)	number	Number of footnote.

0x70 pair			footnote_offsets
Offset	Type	Value	Information
00	footo(n)	offset	Offset to footnotes text. $n=size/4$

Section Info

0x88 pair			section_info
Offset	Type	Value	Information
00	texto(n+1)	offset	
xx	struct(n)	section_info_entry	

section_info_entry			
Offset	Type	Value	Information
00	word		
02	fileo	section_entry	
06	word		
08	long		

section_entry			
Offset	Type	Value	Information
00	wstr	secf	format

Paragraph Info

0x90 pair			par_info
Offset	Type	Value	Information
00	texto(n+1)	offset	Offset to paragraph
xx	struct(n)	par_info_entry	

par_info_entry			
Offset	Type	Value	Information
00	word		unknown
02	word	style	Style number of paragraph

Destinations

0xb0 pair			dest_o
Offset	Type	Value	Information
00	desto()		

Character Formats

0xb8 pair			charf_block_info
If charf_blocko_num equals not n, this table here is invalid. Then char format blockos are charf_blocko_num consequent blockos starting with blocko number charf_blocko_first. (See Header -> 0x18a, 0x18e)			
Offset	Type	Value	Information
00	texto(n+1)		First offset of character format blocko #i.
xx	blocko(n)		Character format offsets.

charf_block			
Offset	Type	Value	Information
00	fileo(n+1)	text_o	Offset to text.
xx	struct(n)	charf_info	
yy	struct(n)	charf_style	yy[i] = fod_o[i] ? fod_o*2 : default;
1ff	byte	n	

charf_info			
Offset	Type	Value	Information
00	byte	fod_offset	

charf_style			
Offset	Type	Value	Information
00	bstr	format	Character format string. See format codes.

Paragraph Formats

0xc0 pair		parf_block_info	
If parf_blocko_num equals not n, this table here is invalid. Then paragraph format blockos are parf_blocko_num consequent blockos starting with blocko number parf_blocko_first. (See Header -> 0x18c, 0x190)			
Offset	Type	Value	Information
00	texto (n+1)		First offset of paragraph format blocko #i.
xx	blocko (n)	parf_blocko	Offsets to paragraph format blocks.

parf_block			
Offset	Type	Value	Information
00	fileo (n+1)	text_o	Offset to text.
xx	struct (n)	parf_entry	
yy	struct (n)	parf_style	
1ff	byte	n	

parf_entry			
Offset	Type	Value	Information
00	byte	0	default style
		style_o	according parf_style has offset style_o*2
01	struct	parf_info	

parf_info			
Offset	Type	Value	Information
00	byte (6)		unknown

parf_style			
Offset	Type	Value	Information
00	byte	1	Size of following data in words(!)
01	word	parf_style_num	Paragraph style number. Refers to stylesheet_par_entry.style_num?
03	str(2*1-2)	format	Paragraph format codes. See format codes.

Format . Codes

Units				
	twip	point	inch	cm
1 twip =	1	1/20	1/1440	0.00176
1 point =	20	1	1/72	0.03528
1 inch =	1440	72	1	2.54
1 cm =	566.9	28.34	0.3937	1

Format Types				
Format type	Offset	Type	Value	Information
.thing	00	byte	ID	
	01	thing	A thing with id ID.	
distance	00	word	n	A distance of n twips.
option	00	byte	0,1,2,...	
switch	00	byte	0,1	0=off, 1=on

Format Codes					
ID	Offsets	Type	Value	English	Deutsch

00	00	byte	0	unknown	
02	00	.word		unknown	
05	00	.option	type	Paragraph alignment	Absatz Ausrichtung
07	00	.switch			Zeilen nicht trennen
08	00	.switch			Absätze nicht trennen
09	00	.switch			Seitenwechsel oberhalb
0c	00	.byte	size	unknown	
	02	byte(size)		unknown	
0d	00	.option		unknown	
0e	00	.switch			Zeilennummern unterdrücken
0f	00	.word	size	Tabulator definition, size of following structure	
	03	byte	n		
	04	byte	0x53		
	05	distance (n)	pos		Tabulatorpositionen
	xx	word(n)	type		Tabulatortypen
10	00	.distance	n		Einzug rechts
11	00	.distance	n		Einzug links
13	00	.distance	n		Einzug erste Zeile
14	00	.distance	n		Zeilenabstand. n>0: mindestens n n<0: genau -n
	03	switch			off==einfach on==mehrfach
15	00	.distance	n		Abstand vor
16	00	.distance	n		Abstand nach
18	00	.switch		New table cell	Neue Tabellenspalte
19	00	.switch		New table row	Neue Tabellenreihe
1b	00	.distance		unknown	
1c	00	.word		unknown	
1d	00	.byte		unknown	
2c	00	.switch			Silbentrennung aus
33	00	.switch			Absatzkontrolle (1)
44	00	byte	0x44		Einfügen?
	01	byte	4		Something?
	02	fileo	ptr		Ptr to something?

4a	00	.byte(4)		unknown	
50	00	.word	style_num	Select character style style_num	
55	00	.switch		bold	fett
56	00	.switch		italic	kursiv
57	00	.switch		strike through	durchgestrichen
5a	00	.switch		small caps	Kapitälchen
5b	00	.switch		capitals	Großbuchstaben
5c	00	.switch		hidden	verborgen
5d	00	.word	n	Font #n	Zeichensatz #n
5e	00	.switch		underline	unterstrichen
60	00	.distance	n		Laufweite n>0: sperren um n n<0: schmälern um -n
61	00	.distance		unknown	
62	00	.option	color	Foreground Color #color	
63	00	.word	n		Zeichensatz mit n/2 Punkte Größe
65	00	.word	n		n>0: n/2 Punkte höherstellen n<0: -n/2 Punkte tieferstellen
67	00	.option		unknown	
68	00	.option	0,1,2		0==normal 1==hochgestellt 2==tiefgestellt
6b	00	.word	n		n>0: unterschneiden ab n/2 Punkte n<0: -n/2 Punkte ?
75	00	.switch		End of text section mark	
88	00	.byte	n		Spaltennummer {0..}
	02	distance	width		Spaltenbreite
89	00	.byte	n		Spaltennummer {0..}
	02	distance	dist		Spaltenabstand
8a	00	.switch		unknown	
8e	00	.option		unknown	
90	00	.distance		unknown	
99	00	.switch		unknown	
9e	00	.switch			Linie zwischen Spalten
a4	00	.distance			Seitenbreite?

a5	00	.distance			Seitenhöhe?
b4	00	.byte		unknown	
b6	00	.word		unknown	
b8	00	.distance			Halber Abstand zwischen zwei Zellen
b9	00	.switch			Kein Seitenwechsel in Zeile?
ba	00	.byte		unknown	
bb	00	.struct	rborder	Border definition for a whole table row.	
bd	00	.distance		Height of one table row.	Höhe einer Tabellenreihe.
be	00	.byte	size		Table definition, size of following structure. If size is not enough for this structure, missing data will be assumed to have value zero.
	02	byte	0	unknown	
	03	byte	n		
	4	distance (n+1)	pos		Position der horizontalen Tabellenstege
	xx	struct(n)	cborder	n border definitions for n cells.	
bf	00	.byte	size	Table definition, size of following structure	
	02	distance (size/2)	width		Breite der Tabellenzellen?
c0	00	.word		unknown	

Paragraph alignment		
Value	English	Deutsch
0x00	left	links
0x01	center	zentriert
0x02	right	rechts
0x03	justify	block

Tabulator types		
Value	English	Deutsch
0x00	left	Links
0x01	right	Rechts
0x02	decimal	Dezimal
0x03	vertical line	Vertikale Linie
0x04	dots	Punkte
0x08	hyphens	Kleine Striche
0x10	line	Linie

Standard Colors?		
Value	English	Deutsch
0x00		?
0x01		schwarz
0x02		hellblau
0x03		helltürkis
0x04		hellgrün
0x05		hellviolett
0x06		hellrot
0x07		gelb
0x08		weiß
0x09		blau
0x0a		türkis
0x0b		grün
0x0c		violett
0x0d		rot
0x0e		braun
0x0f		grau
0x10		grau

Border Definition For One Cell			
Offset	Type	Value	Information
00		0	unknown
02	distance	top	Thickness of top line.
04	distance	left	
06	distance	bottom	
08	distance	right	

Border Definition For A Whole Table Row			
Offset	Type	Value	Information
00	distance	outer_top?	Thickness of line?
02	distance	outer_left?	
04	distance	outer_bottom?	
06	distance	outer_right?	
08	distance	inner_horizontal?	
0a	distance	inner_vertical?	

Font Info

0xd0 pair			font_info
Offset	Type	Value	Information
00	word	size	Size of structure.
02	struct()	font_entry	

font_entry			
Offset	Type	Value	Information
00	byte	size	Size of following structure.
01	byte	family	Font family.
02	byte		unknown
03	byte		unknown
04	byte	charset	Character set identifier.
05	byte	altname_o	Offset to alternative font name.
06	zstr	name	Name of font
06 + altname_o	empty		
	zstr	altname	Alternative font name.

Font Family		
Bits	Value	Information
0..1	00	Default pitch
	01	Fixed pitch
	02	Variable pitch
2..3		unknown
4..7	00	nil, unknown or default
	10	Roman
	20	Swiss
	30	Modern
	40	Script
	50	Decor
	other	?

Font Character Sets	
Value	Information
00	ANSI
02	SYMBOL
80	SHIFTJIS
88	unknown
a1	GREEK
a2	TURKISH
b1	HEBREW
b2	ARABICSIMPLIFIED
b3	ARABICTRADITIONAL
b4	ARABICUSER
b5	HEBREWUSER
cc	CYRILLIC
ee	EASTERNEUROPE
fe	PC437
ff	OEM

Field Info

0xd8 pair			field_info
Offset	Type	Value	Information
00	texto(n+1)		Offset to field mark. The first byte at offset equals to the value of the ID byte of the according struct. If ID of the next field mark is less or equal to current ID, between offset+1 of the current field mark and the offset of the following field mark stands a text.
xx	struct(n)	field_cmd	Command token. s.b.
		field_name	Defined by system or userEss
		field_end	unknown

0xe0 pair			dest_field_info
Offset	Type	Value	Information
00	desto(n+1)		Like field_info. Offsets here don't refer to body text, but to destination text.
xx	struct(n)	field_cmd	Command token. s.b.
		field_name	Defined by system or userEss
		field_end	unknown

field_cmd			
Offset	Type	Value	Information
00	byte	0x13	ID
01	byte	code	

field_name			
Offset	Type	Value	Information
00	byte	0x14	ID
		0x94	
01	byte	0xff	unknown

field_end			
Offset	Type	Value	Information
00	byte	0x15	ID
		0x95	
01	byte	0x40	unknown
		0x80	
		0xc0	

Field Codes			
Value	English	Deutsch	Parameter example

03	REF	REF	Figure1
06	SET	BESTIMMEN	
07	IF	WENN	
08	INDEX	INDEX	\c "2"
0a	STYLEREF	FVREF	"Gliederung 1"
0c	SEQ	SEQ	
0d	TOC	VERZEICHNIS	\o "1-3"
0e	INFO	INFO	
0f	TITLE	TITEL	
10	SUBJECT	THEMA	
11	AUTHOR	AUTOR	
12	KEYWORDS	SCHLÜSSEL	
13	COMMENTS	KOMMENTAR	
14	LASTSAVEDBY	GESPEICHERTVON	
15	CREATEDATE	ERSTELLDAT	
16	SAVEDATE	SPEICHERDAT	
17	PRINTDATE	DRUCKDAT	
18	REVNUM	ÜBERARBEITUNGSNUMMER	
1a	NUMPAGES	ANZSEITEN	
1b	NUMWORDS	ANZWÖRTER	
1c	NUMCHARS	ANZZEICHEN	
1d	FILENAME	DATEINAME	
1e	TEMPLATE	DOKVORLAGE	
1f	DATE	AKTUALDAT	
20	TIME	ZEIT	\@ "tt.MM.jjjj"
21	PAGE	SEITE	3
22	=	=	
23	QUOTE	ANGEBEN	
25	PAGEREF	SEITENREF	
26	ASK	FRAGE	
27	FILLIN	EINGEBEN	
29	NEXT	NÄCHSTER	
2a	NEXTIF	NWENN	
2b		ÜBERSPRINGEN	
2c	MERGEREC	DATENSATZ	
30	PRINT	DRUCK	
31	EQ	FORMEL	

32	GOTOBUTTON	GEHEZU	
33	MACROBUTTON	MAKROSCHALTFLÄCHE	
34	AUTONUMOUT	AUTONRGLI	
35	AUTONUMLGL	AUTONRDEZ	
36	AUTONUM	AUTONR	
37		IMPORT	
38	LINK	VERKNÜPFUNG	
39	SYMBOL	SONDZEICHEN	224 \f "Wingdings" \s 10
3a	EMBED	EINBETTEN	Paint.Picture Object1
3b	MERGEFIELD	SERIENDRUCKFELD	
3c	USERNAME	BENUTZERNAME	
3d	USERINITIALS	BENUTZERINITIALEN	
3e	USERADDRESS	BENUTZERADR	
41	SECTION	ABSCHNITT	
42	SECTIONPAGES	ABSCHNITTSEITEN	
43	INCLUDEPICTURE	EINFÜGENGRAFIK	C:\\WORD\\BILD
44	INCLUDETEXT	EINFÜGENTEXT	
45	FILESIZE	DATEIGRÖSSE	
46		FORMULARTEXT	
47	FORMCHECKBOX	FORMULARKONTROLLFELD	
48	NOTEREF	FUSSENDNOTEREF	
4b	MERGESEQ	SERIENDRUCKSEQ	
4e	DATABASE	DATENBANK	
4f	AUTOTEXT	AUTOTEXT	
50	COMPARE	VERGLEICH	
53		FORMULARDROPDOWN	
54	ADVANCE	VERSETZEN	

Anchors

0x100 pair			anchor_names
00	word	size	Size of structure.
02	bstr()	Names	Anchor names.

0x108 pair			anchor_begin_o
00	texto(n+1)	begin	Anchor begin offset.
xx	long(n)	index	Anchor end index. Refers to anch_end_o.

0x110 pair			anchor_end_o
00	texto()	end	Anchor end offset.

Macro Info

Note: A lot of information I did not yet understand fully or I simply found no time to include it into this specification. Nevertheless a glance into "word6/macrolib.pl" might give you further hints. Especially it could explain to you, how the macro code itself is structured.

0x118 pair			macro_info
Offset	Type	Value	Information
00	byte	0xff	Start ID
01	struct()	empty	
		macro_info	
		menu_info	
		macro_extnames	
		macro_intnames	
		other_info	unknown
xx	byte	0x40	End ID

macro_info			
Offset	Type	Value	Information
00	byte	01	Macro Info ID
01	word	n	Number of entries
03	struct(n)	macro_info_entry	

macro_info_entry			
Offset	Type	Value	Information
00	byte	version	Version of WordBasic?
01	byte	key	XOR mask used on macro text
02	word	intname_i	"internal name", index on macro_intnames.
04	word	extname_i	"external name", raw index on macro_extnames. Index is yielded by taking the index of the ordered list of extname_i, xname_i and menu_entry->extname_i. Value 0xffff means: not defined.
06	word	xname_i	raw index on macro_extnames
08	long		unknown
0c	long	code_len	length of macro text
10	long	code_state	State of macro?
14	file	code	Offset to macro text

menu_info			
Offset	Type	Value	Information
00	byte	05	Menu Info ID
01	word	n	Number of entries
03	struct (n)	menu_info_entry	

menu_info_entry			
Offset	Type	Value	Information
00	word	context	Context of menu entry. (To be explained)
02	word	menu	Menu number. Says in which menu the entry shall be available.
04	word	extname_i	"external name", raw index on macro_extnames
06	word	uk	Unknow value. Always 2?
08	word	intname_i	index on macro_intnames.
10	word	pos	Menu position of menu entry.

macro_extnames			
Offset	Type	Value	Information
00	byte	0x10	External Names ID
01	word	size	Size of entry
03	struct ()	macro_extnames_entry	

macro_extnames_entry			
Offset	Type	Value	Information
00	bstr	string	External names and macro descriptions.
xx	word	numref	Counts, how often this entry is referred to.

macro_intnames			
Offset	Type	Value	Information
00	byte	0x11	Internal Names ID
01	word	n	Number of entries.
03	struct (n)	macro_intnames_entry	

macro_intnames_entry			
Offset	Type	Value	Information
00	word	index	Number to identify this entry.
02	bstr	name	Name string.
xx	byte		unknown

PrinterEss Info

0x130 pair			printer_info
Offset	Type	Value	Information
00	zstr	name	Name of printer?
xx	zstr	connection	Printer connection?
yy	zstr	driver	Printer driver?

Summary Information

0x150 pair			summary_info
Offset	Type	Value	Information
14	long	create_dtm	
18	long	lastsave_dtm	
1c	long	lastprinted_dtm	
20	word	revision	
22	long		
26	long	wordcount	
2a	long	charcount	
2e	word	pagecount	
30	long		(docsuminfo->6)
34	word		
36	word		
38	long		(docsuminfo->5)
3c	long		
40	long		
44	word		
54	long		Word 7 only.

0x158 pair			summary_strings
Offset	Type	Value	Information
00	word	size	Size of structure.
02	byte	0	unknown
03	byte	0	unknown
04	bstr	template	
xx	bstr	keywords	
xx	bstr	short	
xx	bstr	comments	
xx	bstr	authress	
xx	bstr	previous authress	

Fast Save Information

0x160 pair			fastsave_info
Offset	Type	Value	Information
00	struct()	fastsave_empty	
		fastsave_charf	
		fastsave_parf	

fastsave_empty			
Offset	Type	Value	Information
00	byte	0	id
01	byte		unknown

fastsave_charf			
Offset	Type	Value	Information
00	byte	1	id
01	word	len	len of char format style
03	str(len)	charf	char format style

fastsave_parf			
Offset	Type	Value	Information
00	byte	2	
01	word	len	
03	word		unknown
05	texto(n)		
xx	struct(n-1)	fastsave_parf_entry	

fastsave_parf_entry			
Offset	Type	Value	Information
00	word		
02	fileo		
06	word	n	if odd n: fastsave_charf index = (n-1)/2

Document History Information

0x29a pair			history_info
Offset	Type	Value	Information
00	word	size	Size of this structure in bytes.
02	struct()	history_entry	

history_entry			
Offset	Type	Value	Information
00	bstr	authress	
n+1	bstr	path	

[Back to Laola homepage.](#)

TIFF™

Revision 6.0

Final — June 3, 1992

Aldus Developers Desk

Aldus Corporation
411 First Avenue South
Seattle, WA 98104-2871

CompuServe: GO ALDSVC, Message Section #10

Applelink: Aldus Developers Icon

For a copy of the TIFF 6.0 specification, call (206) 628-6593.

If you have questions about the contents of this specification, see page 8.

Copyright

© 1986-1988, 1992 Aldus Corporation. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage and the Aldus copyright notice appears. If the majority of the document is copied or redistributed, it must be distributed verbatim, without repagination or reformatting. To copy otherwise requires specific permission from the Aldus Corporation.

Licenses and Trademarks

Aldus and PageMaker are registered trademarks and TIFF is a trademark of Aldus Corporation. Apple and Macintosh are registered trademarks of Apple Computer, Inc. MS-DOS is a registered trademark of Microsoft Corporation. UNIX is a trademark of Bell Laboratories. CompuServe is a registered trademark of CompuServe Inc. PostScript is a registered trademark of Adobe Systems Inc. and all references to PostScript in this document are references to either the PostScript interpreter or language. Kodak and PhotoYCC are trademarks of Eastman Kodak Company.

Rather than put a trademark symbol in every occurrence of other trademarked names, we state that we are using the names only in an editorial fashion, and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Acknowledgments

This specification is the result of much hard work by many people.

Some of the sections in Part 2 were written by a number of outside contributors:

Ed Beeman, Hewlett Packard

Nancy Cam, Silicon Graphics

Dennis Hamilton, Xerox

Eric Hamilton, C-Cube

Sam Leffler, Silicon Graphics

Chris and Dan Sears

Other primary reviewers and TAC meeting participants include representatives from Apple, Camex, Crosfield, Digital Optics Limited, Frame, IBM, Interleaf, Island Graphics, Kodak, Linotype-Hell, Quark, Sun Microsystems, Time Arts, US West, and Wang. Many thanks to all for lending their time and talents to this effort.

No document this large can completely satisfy everyone, but we have all worked hard to strike an effective balance between power and simplicity, between formality and approachability, and between flexibility and constraints.

Production Notes

This document was created electronically using Aldus PageMaker® 4.2.

Contents

Introduction	4
<i>About this Specification</i>	<i>4</i>
<i>Revision Notes</i>	<i>6</i>
<i>TIFF Administration</i>	<i>8</i>
<i>Information and Support</i>	<i>8</i>
<i>Private Fields and Values</i>	<i>8</i>
<i>Submitting a Proposal</i>	<i>9</i>
<i>The TIFF Advisory Committee</i>	<i>9</i>
<i>Other TIFF Extensions</i>	<i>9</i>
Part 1: Baseline TIFF	11
<i>Section 1: Notation</i>	<i>12</i>
<i>Section 2: TIFF Structure</i>	<i>13</i>
<i>Section 3: Bilevel Images</i>	<i>17</i>
<i>Section 4: Grayscale Images</i>	<i>22</i>
<i>Section 5: Palette-color Images</i>	<i>23</i>
<i>Section 6: RGB Full Color Images</i>	<i>24</i>
<i>Section 7: Additional Baseline TIFF Requirements</i>	<i>26</i>
<i>Section 8: Baseline Field Reference Guide</i>	<i>28</i>
<i>Section 9: PackBits Compression</i>	<i>42</i>
<i>Section 10: Modified Huffman Compression</i>	<i>43</i>
Part 2: TIFF Extensions	48
<i>Section 11: CCITT Bilevel Encodings</i>	<i>49</i>
<i>Section 12: Document Storage and Retrieval</i>	<i>55</i>
<i>Section 13: LZW Compression</i>	<i>57</i>
<i>Section 14: Differencing Predictor</i>	<i>64</i>
<i>Section 15: Tiled Images</i>	<i>66</i>
<i>Section 16: CMYK Images</i>	<i>69</i>
<i>Section 17: HalftoneHints</i>	<i>72</i>
<i>Section 18: Associated Alpha Handling</i>	<i>77</i>
<i>Section 19: Data Sample Format</i>	<i>80</i>
<i>Section 20: RGB Image Colorimetry</i>	<i>82</i>
<i>Section 21: YCbCr Images</i>	<i>89</i>
<i>Section 22: JPEG Compression</i>	<i>95</i>
<i>Section 23: CIE L*a*b* Images</i>	<i>110</i>
Part 3: Appendices	116
<i>Appendix A: TIFF Tags Sorted by Number</i>	<i>117</i>
<i>Appendix B: Operating System Considerations</i>	<i>119</i>
Index	120

Introduction

About this Specification

This document describes TIFF, a tag-based file format for storing and interchanging raster images.

History

The first version of the TIFF specification was published by Aldus Corporation in the fall of 1986, after a series of meetings with various scanner manufacturers and software developers. It did not have a revision number but should have been labeled Revision 3.0 since there were two major earlier draft releases.

Revision 4.0 contained mostly minor enhancements and was released in April 1987. Revision 5.0, released in October 1988, added support for palette color images and LZW compression.

Scope

TIFF describes image data that typically comes from scanners, frame grabbers, and paint- and photo-retouching programs.

TIFF is not a printer language or page description language. The purpose of TIFF is to describe and store raster image data.

A primary goal of TIFF is to provide a rich environment within which applications can exchange image data. This richness is required to take advantage of the varying capabilities of scanners and other imaging devices.

Though TIFF is a rich format, it can easily be used for simple scanners and applications as well because the number of required fields is small.

TIFF will be enhanced on a continuing basis as new imaging needs arise. A high priority has been given to structuring TIFF so that future enhancements can be added without causing unnecessary hardship to developers.

Features

- TIFF is capable of describing bilevel, grayscale, palette-color, and full-color image data in several color spaces.
- TIFF includes a number of compression schemes that allow developers to choose the best space or time tradeoff for their applications.
- TIFF is not tied to specific scanners, printers, or computer display hardware.
- TIFF is portable. It does not favor particular operating systems, file systems, compilers, or processors.
- TIFF is designed to be extensible—to evolve gracefully as new needs arise.
- TIFF allows the inclusion of an unlimited amount of private or special-purpose information.

Revision Notes

This revision replaces TIFF Revision 5.0.

Paragraphs that contain new or substantially-changed information are shown in italics.

New Features in Revision 6.0

Major enhancements to TIFF 6.0 are described in Part 2. They include:

- CMYK image definition
- A revised RGB Colorimetry section.
- YCbCr image definition
- CIE L*a*b* image definition
- Tiled image definition
- JPEG compression

Clarifications

- The LZW compression section more clearly explains when to switch the coding bit length.
- The interaction between Compression=2 (CCITT Huffman) and PhotometricInterpretation was clarified.
- The data organization of uncompressed data (Compression=1) when BitsPerSample is greater than 8 was clarified. See the Compression field description.
- The discussion of CCITT Group 3 and Group 4 bilevel image encodings was clarified and expanded, and Group3Options and Group4Options fields were renamed T4Options and T6Options. See Section 11.

Organizational Changes

- To make the organization more consistent and expandable, appendices were transformed into numbered sections.
- The document was divided into two parts—Baseline and Extensions—to help developers make better and more consistent implementation choices. Part 1, the Baseline section, describes those features that all general-purpose TIFF readers should support. Part 2, the Extensions section, describes a number of features that can be used by special or advanced applications.
- An index and table of contents were added.

Changes in Requirements

- To illustrate a Baseline TIFF file earlier in the document, the material from Appendix G (“TIFF Classes”) in Revision 5 was integrated into the main body of the specification. As part of this integration, the TIFF Classes terminology was replaced by the more monolithic Baseline TIFF terminology. The intent was to further encourage all mainstream TIFF readers to support the Baseline TIFF requirements for bilevel, grayscale, RGB, and palette-color images.
- Due to licensing issues, LZW compression support was moved out of the “Part 1: Baseline TIFF” and into “Part 2: Extensions.”
- Baseline TIFF requirements for bit depths in palette-color images were weakened a bit.

Changes in Terminology

In previous versions of the specification, the term “tag” referred both to the identifying number of a TIFF field and to the entire field. In this version, the term “tag” refers only to the identifying number. The term “field” refers to the entire field, including the value.

Compatibility

Every attempt has been made to add functionality in such a way as to minimize compatibility problems with files and software that were based on earlier versions of the TIFF specification. The goal is that TIFF files should never become obsolete and that TIFF software should not have to be revised more frequently than absolutely necessary. In particular, Baseline TIFF 6.0 files will generally be readable even by older applications that assume TIFF 5.0 or an earlier version of the specification.

However, TIFF 6.0 files that use one of the major new extensions, such as a new compression scheme or color space, will not be successfully read by older software. In such cases, the older applications must gracefully give up and refuse to import the image, providing the user with a reasonably informative message.

TIFF Administration

Information and Support

The most recent version of the TIFF specification in PostScript format is available on CompuServe ("Go ALDSVC", Library 10) and on AppleLink (Aldus Developers Icon). Sample TIFF files and other TIFF developer information can also be found at these locations.

The Aldus CompuServe forum (Go ALDSVC) can also be used to post messages to other TIFF developers, enabling developers to help each other.

Because of the tremendous growth in the usage of TIFF, Aldus is no longer able to provide a general consulting service for TIFF implementors. TIFF developers are encouraged to study sample TIFF files, read TIFF documentation thoroughly, and work with developers of other products that are important to you.

Most companies that use TIFF can answer questions about support for TIFF in their products. Contact the appropriate product manager or developer support service group.

If you are an experienced TIFF developer and are interested in contract programming for other developers, please contact Aldus. Aldus can give your name to others that might need your services.

Private Fields and Values

An organization might wish to store information meaningful to only that organization in a TIFF file. Tags numbered 32768 or higher, sometimes called private tags, are reserved for that purpose.

Upon request, the TIFF administrator (the Aldus Developers Desk) 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 or less. You do not need to tell the TIFF administrator or anyone else what you plan 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 one or more enumerated values for a particular field (Compression, in our example), to avoid possible conflicts.

Tags and values allocated in the private number range are not prohibited from being included in a future revision of this specification. Several such instances exist in the TIFF specification.

Do not choose your own tag numbers. Doing so could cause serious compatibility problems in the future.

If you need more than 5 or 10 tags, Aldus suggests that you reserve a single private tag, define it as a LONG, and use its value as a pointer (offset) to a private IFD or other data structure of your choosing. Within that IFD, you can use whatever tags you want, since no one else will know that it is an IFD unless you tell them. This gives you some 65,000 private tags.

Submitting a Proposal

Any person or group that wants to propose a change or addition to the TIFF specification should prepare a proposal that includes the following information:

- Name of the person or group making the request, and your affiliation.
- The reason for the request.
- A list of changes exactly as you propose that they appear in the specification. Use inserts, callouts, or other obvious editorial techniques to indicate areas of change, and number each change.
- Discussion of the potential impact on the installed base.
- A list of contacts outside your company that support your position. Include their affiliation.

Please send your proposal to Internet address: `tiff-input@aldus.com`. (From AppleLink, you can send to: `tiff-input@aldus.com@internet#`. From CompuServe, you can send to: `>INTERNET:tiff-input@aldus.com`.) Do not send TIFF implementation questions to this address; see above for Aldus Developers Desk TIFF support policies.

The TIFF Advisory Committee

The TIFF Advisory Committee is a working group of TIFF experts from a number of hardware and software manufacturers. It was formed in the spring of 1991 to provide a forum for debating and refining proposals for the 6.0 release of the TIFF specification. It is not clear if this will be an ongoing group or if it will go into a period of hibernation until pressure builds for another major release of the TIFF specification.

If you are a TIFF expert and think you have the time and interest to work on this committee, contact the Aldus Developers Desk for further information. For the TIFF 6.0 release, the group met every two or three months, usually on the west coast of the U.S. Accessibility via Internet e-mail (or AppleLink or CompuServe, which have gateways to the Internet) is a requirement for membership, since that has proven to be an invaluable means for getting work done between meetings.

Other TIFF Extensions

The Aldus TIFF sections on CompuServe and AppleLink will contain proposed extensions from Aldus and other companies that are not yet approved by the TIFF Advisory Committee.

Many of these proposals will never be approved or even considered by the TIFF Advisory Committee, especially if they represent specialized uses of TIFF that do

not fall within the domain of publishing or general graphics or picture interchange. Use them at your own risk; it is unlikely that these features will be widely supported. And if you do write files that incorporate these extensions, be sure to not call them TIFF files or to mark them in some way so that they will not be confused with mainstream TIFF files.

Aldus will provide a place on Compuserve and Applelink for storing such documents. Contact the Aldus Developers Desk for instructions. We recommend that all submissions be in the form of simple text or portable PostScript form that can be downloaded to any PostScript printer in any computing environment.

If a non-Aldus contact name is listed, please use that contact rather than Aldus for submitting requests for future enhancements to that extension.

Part 1: Baseline TIFF

The TIFF specification is divided into two parts. Part 1 describes *Baseline TIFF*. Baseline TIFF is the core of TIFF, the essentials that all mainstream TIFF developers should support in their products.

Section 1: Notation

Decimal and Hexadecimal

Unless otherwise noted, all numeric values in this document are expressed in decimal. (“H” is appended to hexadecimal values.)

Compliance

Is and *shall* indicate mandatory requirements. All compliant writers and readers must meet the specification.

Should indicates a recommendation.

May indicates an option.

Features designated ‘not recommended for general data interchange’ are considered extensions to Baseline TIFF. Files that use such features shall be designated “Extended TIFF 6.0” files, and the particular extensions used should be documented. A Baseline TIFF 6.0 reader is not required to support any extensions.

Section 2: TIFF Structure

TIFF is an image file format. In this document, a *file* is defined to be a sequence of 8-bit bytes, where the bytes are numbered from 0 to N. The largest possible TIFF file is 2^{32} bytes in length.

A TIFF file begins with an 8-byte *image file header* that points to an *image file directory (IFD)*. An image file directory contains information about the image, as well as pointers to the actual image data.

The following paragraphs describe the image file header and IFD in more detail.

See Figure 1.

Image File Header

A TIFF file begins with an 8-byte image file header, containing the following information:

Bytes 0-1: The byte order used within the file. Legal values are:

“II” (4949.H)

“MM” (4D4D.H)

In the “II” format, byte order is always from the least significant byte to the most significant byte, for both 16-bit and 32-bit integers. This is called *little-endian* byte order. In the “MM” format, byte order is always from most significant to least significant, for both 16-bit and 32-bit integers. This is called *big-endian* byte order.

Bytes 2-3: An arbitrary but carefully chosen number (42) that further identifies the file as a TIFF file.

The byte order depends on the value of Bytes 0-1.

Bytes 4-7: The offset (in bytes) of the first IFD. 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 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 TIFF file. The first byte of the file has an offset of 0.

Figure 1

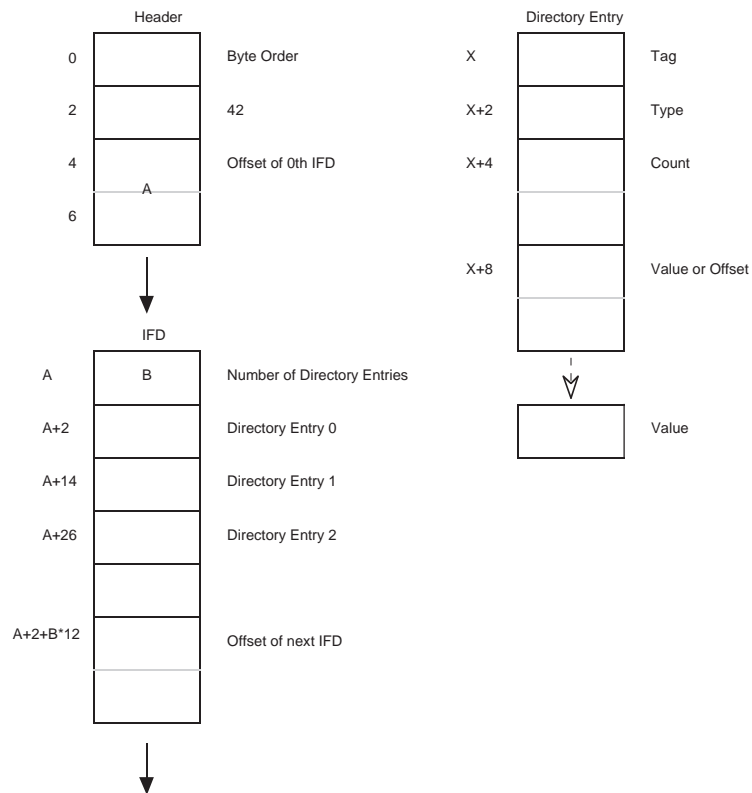


Image File Directory

An *Image File Directory (IFD)* consists of a 2-byte count of the number of directory 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 IFD (or 0 if none). (Do not forget to write the 4 bytes of 0 after the last IFD.)

There must be at least 1 IFD in a TIFF file and each IFD must have at least one entry.

See Figure 1.

IFD Entry

Each 12-byte IFD entry has the following format:

Bytes 0-1 The Tag that identifies the field.

Bytes 2-3 The field Type.

Bytes 4-7 The number of values, *Count* of the indicated Type.

Bytes 8-11 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 anywhere in the file, even after the image data.

IFD Terminology

A *TIFF field* is a logical entity consisting of TIFF tag and its value. This logical concept is implemented as an *IFD Entry*, plus the actual value if it doesn't fit into the value/offset part, the last 4 bytes of the IFD Entry. The terms *TIFF field* and *IFD entry* are interchangeable in most contexts.

Sort Order

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. The Values to which directory entries point need not be in any particular order in the file.

Value/Offset

To save time and space the Value Offset contains the Value instead of pointing to the Value if and only if the Value fits into 4 bytes. If the Value is shorter than 4 bytes, it is left-justified within the 4-byte Value Offset, i.e., stored in the lower-numbered bytes. Whether the Value fits within 4 bytes is determined by the Type and Count of the field.

Count

Count—called *Length* in previous versions of the specification—is the number of values. Note that Count is not the total number of bytes. For example, a single 16-bit word (SHORT) has a Count of 1; not 2.

Types

The field types and their sizes are:

1 = BYTE	8-bit unsigned integer.
2 = ASCII	8-bit byte that contains a 7-bit ASCII code; the last byte must be NUL (binary zero).
3 = SHORT	16-bit (2-byte) unsigned integer.
4 = LONG	32-bit (4-byte) unsigned integer.
5 = RATIONAL	Two LONGs: the first represents the numerator of a fraction; the second, the denominator.

The value of the Count part of an ASCII field entry includes the NUL. If padding is necessary, the Count does not include the pad byte. Note that there is no initial “count byte” as in Pascal-style strings.

Any ASCII field can contain multiple strings, each terminated with a NUL. A single string is preferred whenever possible. The Count for multi-string fields is the number of bytes in all the strings in that field plus their terminating NUL bytes. Only one NUL is allowed between strings, so that the strings following the first string will often begin on an odd byte.

The reader must check the type to verify that it contains an expected value. TIFF currently allows more than 1 valid type for some fields. For example, ImageWidth and ImageLength are usually specified as having type SHORT. But images with more than 64K rows or columns must use the LONG field type.

TIFF readers should accept BYTE, SHORT, or LONG values for any unsigned integer field. This allows a single procedure to retrieve any integer value, makes reading more robust, and saves disk space in some situations.

In TIFF 6.0, some new field types have been defined:

6 = SBYTE	An 8-bit signed (twos-complement) integer.
7 = UNDEFINED	An 8-bit byte that may contain anything, depending on the definition of the field.
8 = SSHORT	A 16-bit (2-byte) signed (twos-complement) integer.
9 = SLONG	A 32-bit (4-byte) signed (twos-complement) integer.
10 = SRATIONAL	Two SLONG's: the first represents the numerator of a fraction, the second the denominator.
11 = FLOAT	Single precision (4-byte) IEEE format.
12 = DOUBLE	Double precision (8-byte) IEEE format.

These new field types are also governed by the byte order (II or MM) in the TIFF header.

Warning: It is possible that other TIFF field types will be added in the future. Readers should skip over fields containing an unexpected field type.

Fields are arrays

Each TIFF field has an associated Count. This means that all fields are actually one-dimensional arrays, even though most fields contain only a single value.

For example, to store a complicated data structure in a single private field, use the UNDEFINED field type and set the Count to the number of bytes required to hold the data structure.

Multiple Images per TIFF File

There may be more than one IFD in a TIFF file. Each IFD defines a *subfile*. One potential use of subfiles is to describe related images, such as the pages of a facsimile transmission. A Baseline TIFF reader is not required to read any IFDs beyond the first one.

Section 3: Bilevel Images

Now that the overall TIFF structure has been described, we can move on to filling the structure with actual fields (tags and values) that describe raster image data.

To make all of this clearer, the discussion will be organized according to the four Baseline TIFF image types: bilevel, grayscale, palette-color, and full-color images. This section describes bilevel images.

Fields required to describe bilevel images are introduced and described briefly here. Full descriptions of each field can be found in Section 8.

Color

A bilevel image contains two colors—black and white. TIFF allows an application to write out bilevel data in either a white-is-zero or black-is-zero format. The field that records this information is called *PhotometricInterpretation*.

PhotometricInterpretation

Tag = 262 (106.H)

Type = SHORT

Values:

- 0 = **WhiteIsZero**. For bilevel and grayscale images: 0 is imaged as white. The maximum value is imaged as black. This is the normal value for *Compression=2*.
- 1 = **BlackIsZero**. For bilevel and grayscale images: 0 is imaged as black. The maximum value is imaged as white. If this value is specified for *Compression=2*, the image should display and print reversed.

Compression

Data can be stored either compressed or uncompressed.

Compression

Tag = 259 (103.H)

Type = SHORT

Values:

- 1 = **No compression**, but pack data into bytes as tightly as possible, leaving no unused bits (except at the end of a row). The component values are stored as an array of type **BYTE**. Each scan line (row) is padded to the next **BYTE** boundary.
- 2 = **CCITT Group 3 1-Dimensional Modified Huffman run length encoding**. See

Section 10 for a description of Modified Huffman Compression.

32773 = PackBits compression, a simple byte-oriented run length scheme. See the PackBits section for details.

Data compression applies only to raster image data. All other TIFF fields are unaffected.

Baseline TIFF readers must handle all three compression schemes.

Rows and Columns

An image is organized as a rectangular array of pixels. The dimensions of this array are stored in the following fields:

ImageLength

Tag = 257 (101.H)

Type = SHORT or LONG

The number of rows (sometimes described as *scanlines*) in the image.

ImageWidth

Tag = 256 (100.H)

Type = SHORT or LONG

The number of columns in the image, i.e., the number of pixels per scanline.

Physical Dimensions

Applications often want to know the size of the picture represented by an image. This information can be calculated from ImageWidth and ImageLength given the following resolution data:

ResolutionUnit

Tag = 296 (128.H)

Type = SHORT

Values:

1 = No absolute unit of measurement. Used for images that may have a non-square aspect ratio but no meaningful absolute dimensions.

2 = Inch.

3 = Centimeter.

Default = 2 (inch).

XResolution

Tag = 282 (11A.H)

Type = RATIONAL

The number of pixels per ResolutionUnit in the ImageWidth (typically, horizontal - see Orientation) direction.

YResolution

Tag = 283 (11B.H)

Type = RATIONAL

The number of pixels per ResolutionUnit in the ImageLength (typically, vertical) direction.

Location of the Data

Compressed or uncompressed image data can be stored almost anywhere in a TIFF file. TIFF also supports breaking an image into separate strips for increased editing flexibility and efficient I/O buffering. The location and size of each strip is given by the following fields:

RowsPerStrip

Tag = 278 (116.H)

Type = SHORT or LONG

The number of rows in each strip (except possibly the last strip.)

For example, if ImageLength is 24, and RowsPerStrip is 10, then there are 3 strips, with 10 rows in the first strip, 10 rows in the second strip, and 4 rows in the third strip. (The data in the last strip is not padded with 6 extra rows of dummy data.)

StripOffsets

Tag = 273 (111.H)

Type = SHORT or LONG

For each strip, the byte offset of that strip.

StripByteCounts

Tag = 279 (117.H)

Type = SHORT or LONG

For each strip, the number of bytes in that strip *after any compression*.

Putting it all together (along with a couple of less-important fields that are discussed later), a sample bilevel image file might contain the following fields:

A Sample Bilevel TIFF File

Offset (hex)	Description	Value (numeric values are expressed in hexadecimal notation)
Header:		
0000	Byte Order	4D4D
0002	42	002A
0004	1st IFD offset	00000014
IFD:		
0014	Number of Directory Entries	000C
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 offset	00000000
Values longer than 4 bytes:		
00B6	StripOffsets	Offset0, Offset1, ... Offset187
03A6	StripByteCounts	Count0, Count1, ... Count187
0696	XResolution	0000012C 00000001
069E	YResolution	0000012C 00000001
06A6	Software	“PageMaker 4.0”
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 Bilevel Image Example

- The IFD in this example starts at 14h. It could have started anywhere in the file providing the offset was an even number greater than or equal to 8 (since the TIFF header is always the first 8 bytes of a TIFF file).
- With 16 rows per strip, there are 188 strips in all.
- The example uses a number of optional fields such as DateTime. TIFF readers must safely skip over these fields if they do not understand or do not wish to use the information. Baseline TIFF readers must not require that such fields be present.
- To make a point, this example has highly-fragmented image data. The strips of the image are not in sequential order. The point of this example is to illustrate that strip offsets must not be ignored. Never assume that strip N+1 follows strip N on disk. It is not required that the image data follow the IFD information.

Required Fields for Bilevel Images

Here is a list of required fields for Baseline TIFF bilevel images. The fields are listed in numerical order, as they would appear in the IFD. Note that the previous example omits some of these fields. This is permitted because the fields that were omitted each have a default and the default is appropriate for this file.

TagName	Decimal	Hex	Type	Value
ImageWidth	256	100	SHORT or LONG	
ImageLength	257	101	SHORT or LONG	
Compression	259	103	SHORT	1, 2 or 32773
PhotometricInterpretation	262	106	SHORT	0 or 1
StripOffsets	273	111	SHORT or LONG	
RowsPerStrip	278	116	SHORT or LONG	
StripByteCounts	279	117	LONG or SHORT	
XResolution	282	11A	RATIONAL	
YResolution	283	11B	RATIONAL	
ResolutionUnit	296	128	SHORT	1, 2 or 3

Baseline TIFF bilevel images were called TIFF Class B images in earlier versions of the TIFF specification.

Section 4: Grayscale Images

Grayscale images are a generalization of bilevel images. Bilevel images can store only black and white image data, but grayscale images can also store shades of gray.

To describe such images, you must add or change the following fields. The other required fields are the same as those required for bilevel images.

Differences from Bilevel Images

Compression = 1 or 32773 (*PackBits*). In Baseline TIFF, grayscale images can either be stored as uncompressed data or compressed with the PackBits algorithm.

Caution: PackBits is often ineffective on continuous tone images, including many grayscale images. In such cases, it is better to leave the image uncompressed.

BitsPerSample

Tag = 258 (102.H)

Type = SHORT

The number of bits per component.

Allowable values for Baseline TIFF grayscale images are **4** and **8**, allowing either 16 or 256 distinct shades of gray.

Required Fields for Grayscale Images

These are the required fields for grayscale images (in numerical order):

TagName	Decimal	Hex	Type	Value
ImageWidth	256	100	SHORT or LONG	
ImageLength	257	101	SHORT or LONG	
BitsPerSample	258	102	SHORT	4 or 8
Compression	259	103	SHORT	1 or 32773
PhotometricInterpretation	262	106	SHORT	0 or 1
StripOffsets	273	111	SHORT or LONG	
RowsPerStrip	278	116	SHORT or LONG	
StripByteCounts	279	117	LONG or SHORT	
XResolution	282	11A	RATIONAL	
YResolution	283	11B	RATIONAL	
ResolutionUnit	296	128	SHORT	1 or 2 or 3

Baseline TIFF grayscale images were called TIFF Class G images in earlier versions of the TIFF specification.

Section 5: Palette-color Images

Palette-color images are similar to grayscale images. They still have one component per pixel, but the component value is used as an index into a full RGB-lookup table. To describe such images, you need to add or change the following fields. The other required fields are the same as those for grayscale images.

Differences from Grayscale Images

PhotometricInterpretation = 3 (Palette Color).

ColorMap

Tag = 320 (140.H)

Type = SHORT

N = $3 * (2^{**}BitsPerSample)$

This field defines a Red-Green-Blue color map (often called a lookup table) for palette color images. In a palette-color image, a pixel value is used to index into an RGB-lookup table. For example, a palette-color pixel having a value of 0 would be displayed according to the 0th Red, Green, Blue triplet.

In a TIFF ColorMap, all the Red values come first, followed by the Green values, then the Blue values. In the ColorMap, black is represented by 0,0,0 and white is represented by 65535, 65535, 65535.

Required Fields for Palette Color Images

These are the required fields for palette-color images (in numerical order):

TagName	Decimal	Hex	Type	Value
ImageWidth	256	100	SHORT or LONG	
ImageLength	257	101	SHORT or LONG	
BitsPerSample	258	102	SHORT	4 or 8
Compression	259	103	SHORT	1 or 32773
PhotometricInterpretation	262	106	SHORT	3
StripOffsets	273	111	SHORT or LONG	
RowsPerStrip	278	116	SHORT or LONG	
StripByteCounts	279	117	LONG or SHORT	
XResolution	282	11A	RATIONAL	
YResolution	283	11B	RATIONAL	
ResolutionUnit	296	128	SHORT	1 or 2 or 3
ColorMap	320	140	SHORT	

Baseline TIFF palette-color images were called TIFF Class P images in earlier versions of the TIFF specification.

Section 6: RGB Full Color Images

In an RGB image, each pixel is made up of three components: red, green, and blue. There is no ColorMap.

To describe an RGB image, you need to add or change the following fields and values. The other required fields are the same as those required for palette-color images.

Differences from Palette Color Images

BitsPerSample = 8,8,8. Each component is 8 bits deep in a Baseline TIFF RGB image.

PhotometricInterpretation = 2 (RGB).

There is no **ColorMap**.

SamplesPerPixel

Tag = 277 (115.H)

Type = SHORT

The number of components per pixel. This number is 3 for RGB images, unless extra samples are present. See the ExtraSamples field for further information.

Required Fields for RGB Images

These are the required fields for RGB images (in numerical order):

TagName	Decimal	Hex	Type	Value
ImageWidth	256	100	SHORT or LONG	
ImageLength	257	101	SHORT or LONG	
BitsPerSample	258	102	SHORT	8,8,8
Compression	259	103	SHORT	1 or 32773
PhotometricInterpretation	262	106	SHORT	2
StripOffsets	273	111	SHORT or LONG	
SamplesPerPixel	277	115	SHORT	3 or more
RowsPerStrip	278	116	SHORT or LONG	
StripByteCounts	279	117	LONG or SHORT	
XResolution	282	11A	RATIONAL	
YResolution	283	11B	RATIONAL	
ResolutionUnit	296	128	SHORT	1, 2 or 3

The BitsPerSample values listed above apply only to the main image data. If ExtraSamples are present, the appropriate BitsPerSample values for those samples must also be included.

Baseline TIFF RGB images were called TIFF Class R images in earlier versions of the TIFF specification.

Section 7: Additional Baseline TIFF Requirements

This section describes characteristics required of all Baseline TIFF files.

General Requirements

Options. Where there are options, TIFF writers can use whichever they want. Baseline TIFF readers must be able to handle all of them.

Defaults. TIFF 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 readers must be prepared to handle either situation.

Other fields. TIFF readers must be prepared to encounter fields other than those required in TIFF files. TIFF writers are allowed to write optional fields such as Make, Model, and DateTime, and TIFF readers may use such fields if they exist. TIFF readers must not, however, refuse to read the file if such optional fields do not exist. *TIFF readers must also be prepared to encounter and ignore private fields not described in the TIFF specification.*

‘MM’ and ‘II’ byte order. TIFF readers must be able to handle both byte orders. TIFF writers can do whichever is most convenient or efficient.

Multiple subfiles. TIFF readers must be prepared for multiple images (subfiles) per TIFF file, although they are not required to do anything with images after the first one. TIFF writers are required to write a long word of 0 after the last IFD (to signal that this is the last IFD), as described earlier in this specification.

If multiple subfiles are written, the first one must be the full-resolution image. Subsequent images, such as reduced-resolution images, may be in any order in the TIFF file. If a reader wants to use such images, it must scan the corresponding IFD’s before deciding how to proceed.

TIFF 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 they must delete any subfiles that they aren’t prepared to deal with.
- A similar situation arises with the fields in an IFD. It is unnecessary—and possibly dangerous—for an editor to copy fields it does not understand because the editor might alter the file in a way that is incompatible with the unknown fields.

No Duplicate Pointers. *No data should be referenced from more than one place. TIFF readers and editors are under no obligation to detect this condition and handle it properly. This would not be a problem if TIFF files were read-only enti-*

ties, but they are not. This warning covers both TIFF field value offsets and fields that are defined as offsets, such as StripOffsets.

Point to real data. All strip offsets must reference valid locations. (It is not legal to use an offset of 0 to mean something special.)

Beware of extra components. Some TIFF files may have more components per pixel than you think. A Baseline TIFF reader must skip over them gracefully, using the values of the *SamplesPerPixel* and *BitsPerSample* fields. For example, it is possible that the data will have a *PhotometricInterpretation* of RGB but have 4 *SamplesPerPixel*. See *ExtraSamples* for further details.

Beware of new field types. Be prepared to handle unexpected field types such as floating-point data. A Baseline TIFF reader must skip over such fields gracefully. Do not expect that *BYTE*, *ASCII*, *SHORT*, *LONG*, and *RATIONAL* will always be a complete list of field types.

Beware of new pixel types. Some TIFF files may have pixel data that consists of something other than unsigned integers. If the *SampleFormat* field is present and the value is not 1, a Baseline TIFF reader that cannot handle the *SampleFormat* value must terminate the import process gracefully.

Notes on Required Fields

ImageWidth, ImageLength. Both “SHORT” and “LONG” TIFF field types are allowed and must be handled properly by readers. TIFF writers can use either type. TIFF readers are not required to read arbitrarily large files however. Some readers will give up if the entire image cannot fit into available memory. (In such cases the reader should inform the user about the problem.) Others will probably not be able to handle *ImageWidth* greater than 65535.

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. If the entire image is one strip, the application may run out of memory. Recommendation: Set *RowsPerStrip* such that the size of each strip is about 8K bytes. Do this even for uncompressed data because it is easy for a writer and makes things simpler for readers. Note that extremely wide high-resolution images may have rows larger than 8K bytes; in this case, *RowsPerStrip* should be 1, and the strip will be larger than 8K.

StripOffsets. SHORT or LONG.

StripByteCounts. SHORT or LONG.

XResolution, YResolution. RATIONAL. Note that the X and Y resolutions may be unequal. A TIFF reader must be able to handle this case. Typically, TIFF pixel-editors do not care about the resolution, but applications (such as page layout programs) do care.

ResolutionUnit. SHORT. TIFF readers must be prepared to handle all three values for *ResolutionUnit*.

Section 8: Baseline Field Reference Guide

This section contains detailed information about all the Baseline fields defined in this version of TIFF. A *Baseline field* is any field commonly found in a Baseline TIFF file, whether required or not.

For convenience, fields that were defined in earlier versions of the TIFF specification but are no longer generally recommended have also been included in this section.

New fields that are associated with optional features are not listed in this section. See Part 2 for descriptions of these new fields. There is a complete list of all fields described in this specification in Appendix A, and there are entries for all TIFF fields in the index.

More fields may be added in future versions. Whenever possible they will be added in a way that allows old TIFF readers to read newer TIFF files.

The documentation for each field contains:

- the name of the field
- the Tag number
- the field Type
- the required Number of Values (N); i.e., the Count
- comments describing the field
- the default, if any

If the field does not exist, readers must assume the default value for the field.

Most of the fields described in this part of the document are not required or are required only for particular types of TIFF files. See the preceding sections for lists of required fields.

Before defining the fields, you must understand these basic concepts: A Baseline TIFF *image* is defined to be a two-dimensional array of *pixels*, each of which consists of one or more color *components*. Monochromatic data has one color component per pixel, while RGB color data has three color components per pixel.

The Fields

Artist

Person who created the image.

Tag = 315 (13B.H)

Type = ASCII

Note: some older TIFF files used this tag for storing Copyright information.

BitsPerSample

Number of bits per component.

Tag = 258 (102.H)

Type = SHORT

N = SamplesPerPixel

Note that this field allows a different number of bits per component for each component corresponding to a pixel. For example, RGB color data could use a different number of bits per component for each of the three color planes. Most RGB files will have the same number of BitsPerSample for each component. Even in this case, the writer must write all three values.

Default = 1. See also SamplesPerPixel.

CellLength

The length of the dithering or halftoning matrix used to create a dithered or halftoned bilevel file.

Tag = 265 (109.H)

Type = SHORT

N = 1

This field should only be present if Thresholding = 2

No default. See also Thresholding.

CellWidth

The width of the dithering or halftoning matrix used to create a dithered or halftoned bilevel file. Tag = 264 (108.H)

Type = SHORT

N = 1

No default. See also Thresholding.

ColorMap

A color map for palette color images.

Tag = 320 (140.H)

Type = SHORT

N = 3 * (2**BitsPerSample)

This field defines a Red-Green-Blue color map (often called a lookup table) for palette-color images. In a palette-color image, a pixel value is used to index into an RGB lookup table. For example, a palette-color pixel having a value of 0 would be displayed according to the 0th Red, Green, Blue triplet.

In a TIFF ColorMap, all the Red values come first, followed by the Green values, then the Blue values. The number of values for each color is $2^{*} \text{BitsPerSample}$. Therefore, the ColorMap field for an 8-bit palette-color image would have $3 * 256$ values.

The width of each value 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.

See also PhotometricInterpretation—palette color.

No default. ColorMap must be included in all palette-color images.

Compression

Compression scheme used on the image data.

Tag = 259 (103.H)

Type = SHORT

N = 1

- 1 = No compression, but pack data into bytes as tightly as possible leaving no unused bits except at the end of a row.

If Then the sample values are stored as an array of type:

BitsPerSample = 16 for all samples SHORT

BitsPerSample = 32 for all samples LONG

Otherwise BYTE

Each row is padded to the next BYTE/SHORT/LONG boundary, consistent with the preceding BitsPerSample rule.

If the image data is stored as an array of SHORTs or LONGs, the byte ordering must be consistent with that specified in bytes 0 and 1 of the TIFF file header. Therefore, little-endian format files will have the least significant bytes preceding the most significant bytes, while big-endian format files will have the opposite order.

If the number of bits per component is not a power of 2, and you are willing to give up some space for better performance, use the next higher power of 2. For example, if your data can be represented in 6 bits, set BitsPerSample to 8 instead of 6, and then convert the range of the values from [0,63] to [0,255].

Rows must begin on byte boundaries. (*SHORT boundaries if the data is stored as SHORTs, LONG boundaries if the data is stored as LONGs*).

Some graphics systems require image data rows to be word-aligned or double-word-aligned, and padded to word-boundaries or double-word boundaries. Uncompressed TIFF rows will need to be copied into word-aligned or double-word-aligned row buffers before being passed to the graphics routines in these environments.

- 2 = CCITT Group 3 1-Dimensional Modified Huffman run-length encoding. See Section 10. BitsPerSample must be 1, since this type of compression is defined only for bilevel images.

32773 = PackBits compression, a simple byte-oriented run-length scheme. See Section 9 for details.

Data compression applies only to the image data, pointed to by StripOffsets.

Default = 1.

Copyright

Copyright notice.

Tag = 33432 (8298.H)

Type = ASCII

Copyright notice of the person or organization that claims the copyright to the image. The complete copyright statement should be listed in this field including any dates and statements of claims. For example, “Copyright, John Smith, 19xx. All rights reserved.”

DateTime

Date and time of image creation.

Tag = 306 (132.H)

Type = ASCII

N = 20

The format is: “YYYY:MM:DD HH:MM:SS”, with hours like those on a 24-hour clock, and one space character between the date and the time. The length of the string, including the terminating NUL, is 20 bytes.

ExtraSamples

Description of extra components.

Tag = 338 (152.H)

Type = SHORT

N = m

Specifies that each pixel has m extra components whose interpretation is defined by one of the values listed below. When this field is used, the SamplesPerPixel field has a value greater than the PhotometricInterpretation field suggests.

For example, full-color RGB data normally has SamplesPerPixel=3. If SamplesPerPixel is greater than 3, then the ExtraSamples field describes the meaning of the extra samples. If SamplesPerPixel is, say, 5 then ExtraSamples will contain 2 values, one for each extra sample.

ExtraSamples is typically used to include non-color information, such as opacity, in an image. The possible values for each item in the field's value are:

0 = Unspecified data

1 = Associated alpha data (with pre-multiplied color)

2 = Unassociated alpha data

Associated alpha data is opacity information; it is fully described in Section 21. Unassociated alpha data is transparency information that logically exists independent of an image; it is commonly called a soft matte. Note that including both unassociated and associated alpha is undefined because associated alpha specifies that color components are pre-multiplied by the alpha component, while unassociated alpha specifies the opposite.

By convention, extra components that are present must be stored as the “last components” in each pixel. For example, if `SamplesPerPixel` is 4 and there is 1 extra component, then it is located in the last component location (`SamplesPerPixel-1`) in each pixel.

Components designated as “extra” are just like other components in a pixel. In particular, the size of such components is defined by the value of the `BitsPerSample` field.

With the introduction of this field, TIFF readers must not assume a particular `SamplesPerPixel` value based on the value of the `PhotometricInterpretation` field. For example, if the file is an RGB file, `SamplesPerPixel` may be greater than 3.

The default is no extra samples. This field must be present if there are extra samples.

See also `SamplesPerPixel`, `AssociatedAlpha`.

FillOrder

The logical order of bits within a byte.

Tag = 266 (10A.H)

Type = SHORT

N = 1

- 1 = pixels are arranged within a byte such that pixels with lower column values are stored in the higher-order bits of the byte.

1-bit uncompressed data example: Pixel 0 of a row is stored in the high-order bit of byte 0, pixel 1 is stored in the next-highest bit, ..., pixel 7 is stored in the low-order bit of byte 0, pixel 8 is stored in the high-order bit of byte 1, and so on.

CCITT 1-bit compressed data example: The high-order bit of the first compression code is stored in the high-order bit of byte 0, the next-highest bit of the first compression code is stored in the next-highest bit of byte 0, and so on.

- 2 = pixels are arranged within a byte such that pixels with lower column values are stored in the lower-order bits of the byte.

We recommend that `FillOrder=2` be used only in special-purpose applications. It is easy and inexpensive for writers to reverse bit order by using a 256-byte lookup table. *FillOrder = 2 should be used only when BitsPerSample = 1 and the data is either uncompressed or compressed using CCITT 1D or 2D compression, to avoid potentially ambiguous situations.*

Support for `FillOrder=2` is not required in a Baseline TIFF compliant reader

Default is `FillOrder = 1`.

FreeByteCounts

For each string of contiguous unused bytes in a TIFF file, the number of bytes in the string.

Tag = 289 (121.H)

Type = LONG

Not recommended for general interchange.

See also FreeOffsets.

FreeOffsets

For each string of contiguous unused bytes in a TIFF file, the byte offset of the string.

Tag = 288 (120.H)

Type = LONG

Not recommended for general interchange.

See also FreeByteCounts.

GrayResponseCurve

For grayscale data, the optical density of each possible pixel value.

Tag = 291 (123.H)

Type = SHORT

N = 2**BitsPerSample

The 0th value of GrayResponseCurve corresponds to the optical density of a pixel having a value of 0, and so on.

This field may provide useful information for sophisticated applications, but it is currently ignored by most TIFF readers.

See also GrayResponseUnit, PhotometricInterpretation.

GrayResponseUnit

The precision of the information contained in the GrayResponseCurve.

Tag = 290 (122.H)

Type = SHORT

N = 1

Because optical density is specified in terms of fractional numbers, this field is necessary 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.

- 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`.

For historical reasons, the default is 2. However, for greater accuracy, 3 is recommended.

HostComputer

The computer and/or operating system in use at the time of image creation.

Tag = 316 (13C.H)

Type = ASCII

See also `Make`, `Model`, `Software`.

ImageDescription

A string that describes the subject of the image.

Tag = 270 (10E.H)

Type = ASCII

For example, a user may wish to attach a comment such as “1988 company picnic” to an image.

ImageLength

The number of rows of pixels in the image.

Tag = 257 (101.H)

Type = SHORT or LONG

N = 1

No default. See also `ImageWidth`.

ImageWidth

The number of columns in the image, i.e., the number of pixels per row.

Tag = 256 (100.H)

Type = SHORT or LONG

N = 1

No default. See also `ImageLength`.

Make

The scanner manufacturer.

Tag = 271 (10F.H)

Type = ASCII

Manufacturer of the scanner, video digitizer, or other type of equipment used to generate the image. Synthetic images should not include this field.

See also Model, Software.

MaxSampleValue

The maximum component value used.

Tag = 281 (119.H)

Type = SHORT

N = SamplesPerPixel

This field is not to be used to affect the visual appearance of an image when it is displayed or printed. Nor should this field affect the interpretation of any other field; it is used only for statistical purposes.

Default is $2^{**}(\text{BitsPerSample}) - 1$.

MinSampleValue

The minimum component value used.

Tag = 280 (118.H)

Type = SHORT

N = SamplesPerPixel

See also MaxSampleValue.

Default is 0.

Model

The scanner model name or number.

Tag = 272 (110.H)

Type = ASCII

The model name or number of the scanner, video digitizer, or other type of equipment used to generate the image.

See also Make, Software.

NewSubfileType

A general indication of the kind of data contained in this subfile.

Tag = 254 (FE.H)

Type = LONG

N = 1

Replaces the old SubfileType field, due to limitations in the definition of that field.

NewSubfileType is mainly useful when there are multiple subfiles in a single TIFF file.

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 field 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 are defined as bit flags because they are independent of each other.
- Default is 0.

Orientation

The orientation of the image with respect to the rows and columns.

Tag = 274 (112.H)

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.
- 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.

Support for orientations other than 1 is not a Baseline TIFF requirement.

PhotometricInterpretation

The color space of the image data.

Tag = 262 (106.H)

Type = SHORT

N = 1

- 0 = WhiteIsZero. For bilevel and grayscale images: 0 is imaged as white. $2^{**}\text{BitsPerSample}-1$ is imaged as black. This is the normal value for Compression=2.
- 1 = BlackIsZero. For bilevel and grayscale images: 0 is imaged as black. $2^{**}\text{BitsPerSample}-1$ is imaged as white. 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 components, 0 represents minimum intensity, and $2^{**}\text{BitsPerSample} - 1$ represents maximum intensity. Thus an RGB value of (0,0,0) represents black, and (255,255,255) represents white, assuming 8-bit components. For PlanarConfiguration = 1, the components are stored in the indicated order: first Red, then Green, then Blue. For PlanarConfiguration = 2, the StripOffsets for the component planes are stored in the indicated order: first the Red component plane StripOffsets, then the Green plane StripOffsets, then the Blue plane StripOffsets.
- 3 = Palette color. In this model, a color is described with a single component. The value of the component is used as an index into the red, green and blue curves in the ColorMap field to retrieve an RGB triplet that defines the color. When PhotometricInterpretation=3 is used, ColorMap must be present and 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 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.

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.

The image mask is typically at a higher resolution than the main image, if the main image is grayscale or color so that the edges can be sharp.

There is no default for PhotometricInterpretation, *and it is required*. Do not rely on applications defaulting to what you want.

PlanarConfiguration

How the components of each pixel are stored.

Tag = 284 (11C.H)

Type = SHORT

N = 1

- 1 = *Chunky* format. The component values for each pixel are stored contiguously. The order of the components within the pixel is specified by PhotometricInterpretation. For example, for RGB data, the data is stored as RGBRGBRGB...
- 2 = *Planar* format. The components are stored in separate “component 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 stored in each component plane. For example, RGB data is stored with the Red components in one component plane, the Green in another, and the Blue in another.

PlanarConfiguration=2 is not currently in widespread use and it is not recommended for general interchange. It is used as an extension and Baseline TIFF readers are not required to support it.

If SamplesPerPixel is 1, PlanarConfiguration is irrelevant, and need not be included.

If a row interleave effect is desired, a writer might write out the data as PlanarConfiguration=2—separate sample planes—but break up the planes into multiple strips (one row per strip, perhaps) and interleave the strips.

Default is 1. See also BitsPerSample, SamplesPerPixel.

ResolutionUnit

The unit of measurement for XResolution and YResolution.

Tag = 296 (128.H)

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 arbitrary, it might be better to use dots per inch or dots per centimeter, and to pick XResolution and YResolution so that the aspect ratio is correct and the maximum dimension of the image is about four inches (the “four” is arbitrary.)

- 2 = Inch.
- 3 = Centimeter.

Default is 2.

RowsPerStrip

The number of rows per strip.

Tag = 278 (116.H)

Type = SHORT or LONG

N = 1

TIFF image data is organized into strips for faster random access and efficient I/O buffering.

RowsPerStrip and ImageLength together tell us the number of strips in the entire image. The equation is:

StripsPerImage = floor ((ImageLength + RowsPerStrip - 1) / RowsPerStrip).

StripsPerImage is *not* a field. It is merely a value that a TIFF reader will want to compute because it specifies the number of StripOffsets and StripByteCounts for the image.

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 accept SHORT values.

The default is $2^{32} - 1$, which is effectively infinity. That is, the entire image is one strip.

Use of a single strip is not recommended. 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” value is fairly arbitrary, but seems to work well.

See also ImageLength, StripOffsets, StripByteCounts, TileWidth, TileLength, TileOffsets, TileByteCounts.

SamplesPerPixel

The number of components per pixel.

Tag = 277 (115.H)

Type = SHORT

N = 1

SamplesPerPixel is *usually* 1 for bilevel, grayscale, and palette-color images. SamplesPerPixel is *usually* 3 for RGB images.

Default = 1. See also BitsPerSample, PhotometricInterpretation, *ExtraSamples*.

Software

Name and version number of the software package(s) used to create the image.

Tag = 305 (131.H)

Type = ASCII

See also Make, Model.

StripByteCounts

For each strip, the number of bytes in the strip after compression.

Tag = 279 (117.H)

Type = SHORT or LONG

N = StripsPerImage for PlanarConfiguration equal to 1.

= SamplesPerPixel * StripsPerImage for PlanarConfiguration equal to 2

This tag is required for Baseline TIFF files.

No default.

See also StripOffsets, RowsPerStrip, TileOffsets, TileByteCounts.

StripOffsets

For each strip, the byte offset of that strip.

Tag = 273 (111.H)

Type = SHORT or LONG

N = StripsPerImage for PlanarConfiguration equal to 1.

= SamplesPerPixel * StripsPerImage for PlanarConfiguration equal to 2

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 required field is the only way for a reader to find the image data. (*Unless TileOffsets is used; see TileOffsets.*)

Note that either SHORT or LONG values may 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 accept SHORT values.

For maximum compatibility with operating systems such as MS-DOS and Windows, the StripOffsets array should be less than or equal to 64K bytes in length, and the strips themselves, in both compressed and uncompressed forms, should not be larger than 64K bytes.

No default. See also StripByteCounts, RowsPerStrip, TileOffsets, TileByteCounts.

SubfileType

A general indication of the kind of data contained in this subfile.

Tag = 255 (FF.H)

Type = SHORT

N = 1

Currently defined values are:

- 1 = full-resolution image data
- 2 = reduced-resolution image data
- 3 = a single page of a multi-page image (see the PageNumber field description).

Note that several image types may be found in a single TIFF file, with each subfile described by its own IFD.

No default.

This field is deprecated. The NewSubfileType field should be used instead.

Thresholding

For black and white TIFF files that represent shades of gray, the technique used to convert from gray to black and white pixels.

Tag = 263 (107.H)

Type = SHORT

N = 1

- 1 = No dithering or halftoning has been applied to the image data.
- 2 = An ordered dither or halftone technique has been applied to the image data.
- 3 = A randomized process such as error diffusion has been applied to the image data.

Default is Thresholding = 1. See also CellWidth, CellLength.

XResolution

The number of pixels per ResolutionUnit in the ImageWidth direction.

Tag = 282 (11A.H)

Type = RATIONAL

N = 1

It is not mandatory that the image be actually displayed or 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

The number of pixels per ResolutionUnit in the ImageLength direction.

Tag = 283 (11B.H)

Type = RATIONAL

N = 1

No default. See also XResolution, ResolutionUnit.

Section 9: PackBits Compression

This section describes TIFF compression type 32773, a simple byte-oriented run-length scheme.

Description

In choosing a simple byte-oriented run-length compression scheme, we arbitrarily chose the Apple Macintosh PackBits scheme. It has a good worst case behavior (at most 1 extra byte for every 128 input bytes). For Macintosh users, the toolbox utilities PackBits and UnPackBits 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 is best to encode a 2-byte repeat run as a replicate run except when preceded and followed by a literal run. In that case, it is best to merge the three runs into one literal run. Always encode 3-byte repeats as replicate runs.

That is the essence of the algorithm. Here are some additional rules:

- Pack each row separately. Do not compress across row boundaries.
- The number of uncompressed bytes per row is defined to be $(\text{ImageWidth} + 7) / 8$. 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, encode the remainder of the run as one or more additional replicate runs.

When PackBits data is decompressed, the result should be interpreted as per compression type 1 (no compression).

Section 10: Modified Huffman Compression

This section describes TIFF compression scheme 2, a method for compressing bilevel data based on the CCITT Group 3 1D facsimile compression scheme.

References

- “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.
- “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 section. 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.

See Section 11 for additional CCITT compression options.

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 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.

To ensure that the receiver (decompressor) maintains color synchronization, all data lines begin with a white run-length code word set. If the actual scan line begins with a black run, a white run-length of zero is 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.

An encoded CCITT string is self-photometric, defined in terms of white and black runs. Yet TIFF defines a tag called PhotometricInterpretation that also purports to define what is white and what is black. Somewhat arbitrarily, we adopt the following convention:

The “normal” PhotometricInterpretation for bilevel CCITT compressed data is WhiteIsZero. In this case, the CCITT “white” runs are to be interpreted as white, and the CCITT “black” runs are to be interpreted as black. However, if the PhotometricInterpretation is BlackIsZero, the TIFF reader must reverse the meaning of white and black when displaying and printing the image.

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
6	0010011	26	000011001010
27	0100100	27	000011001011
28	0011000	28	000011001100
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

White run length	Code word	Black run length	Code word
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
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	00000000000

Additional make-up codes

White and Black run length	Make-up code word
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

Part 2: TIFF Extensions

Part 2 contains extensions to Baseline TIFF. TIFF Extensions are TIFF features that may not be supported by all TIFF readers. TIFF creators who use these features will have to work closely with TIFF readers in their part of the industry to ensure successful interchange.

The features described in this part were either contained in earlier versions of the specification, or have been approved by the TIFF Advisory Committee.

Section 11: CCITT Bilevel Encodings

The following fields are used when storing binary pixel arrays using one of the encodings adopted for raster-graphic interchange in numerous CCITT and ISO (International Organization for Standards) recommendations and standards. These encodings are often spoken of as “Group III compression” and “Group IV compression” because their application in facsimile transmission is the most widely known.

For the specialized use of these encodings in storing facsimile-transmission images, further guidelines can be obtained from the TIFF Class F document, available on-line in the same locations as this specification. This document is administered by another organization; paper copies are not available from Aldus.

Compression

Tag = 259 (103.H)

Type = SHORT

N = 1

- 3 = T4-encoding: CCITT T.4 bi-level encoding as specified in section 4, Coding, of CCITT Recommendation T.4: “Standardization of Group 3 Facsimile apparatus for document transmission.” International Telephone and Telegraph Consultative Committee (CCITT, Geneva: 1988).

See the T4Options field for T4-encoding options such as 1D vs 2D coding.

- 4 = T6-encoding: CCITT T.6 bi-level encoding as specified in section 2 of CCITT Recommendation T.6: “Facsimile coding schemes and coding control functions for Group 4 facsimile apparatus.” International Telephone and Telegraph Consultative Committee (CCITT, Geneva: 1988).

See the T6Options field for T6-encoding options such as escape into uncompressed mode to avoid negative-compression cases.

Application in Image Interchange

CCITT Recommendations T.4 and T.6 are specified in terms of the serial bit-by-bit creation and processing of a variable-length binary string that encodes bi-level (black and white) pixels of a rectangular image array. Generally, the encoding schemes are described in terms of bit-serial communication procedures and the end-to-end coordination that is required to gain reliable delivery over inherently unreliable data links. The Group 4 procedures, with their T6-encoding, represent a significant simplification because it is assumed that a reliable communication medium is employed, whether ISDN or X.25 or some other trustworthy transport vehicle. Because image-storage systems and computers achieve data integrity and communication reliability in other ways, the T6-encoding tends to be preferred for imaging applications. When computer storage and retrieval and interchange of facsimile material are of interest, the T4-encodings provide a better match to the

current generation of Group 3 facsimile products and their defenses against data corruption as the result of transmission defects.

Whichever form of encoding is preferable for a given application, there are a number of adjustments that need to be made to account for the capture of the CCITT binary-encoding strings as part of electronically-stored material and digital-image interchange.

PhotometricInterpretation. An encoded CCITT string is self-photometric, defined in terms of white and black runs. Yet TIFF defines a tag called *PhotometricInterpretation* that also purports to define what is white and what is black. Somewhat arbitrarily, we adopt the following convention:

The “normal” *PhotometricInterpretation* for bilevel CCITT compressed data is *WhiteIsZero*. In this case, the CCITT “white” runs are to be interpreted as white, and the CCITT “black” runs are to be interpreted as black. However, if the *PhotometricInterpretation* is *BlackIsZero*, the TIFF reader must reverse the meaning of white and black when displaying and printing the image.

FillOrder. When CCITT encodings are used directly over a typical serial communication link, the order of the bits in the encoded string is the sequential order of the string, bit-by-bit, from beginning to end. This poses the following question: In which order should consecutive blocks of eight bits be assembled into octets (standard data bytes) for use within a computer system? The answer differs depending on whether we are concerned about preserving the serial-transmission sequence or preserving only the format of byte-organized sequences in memory and in stored files.

From the perspective of electronic interchange, as long as a receiver’s reassembly of bits into bytes properly mirrors the way in which the bytes were disassembled by the transmitter, no one cares which order is seen on the transmission link because each multiple of 8 bits is transparently transmitted.

Common practice is to record arbitrary binary strings into storage sequences such that the first sequential bit of the string is found in the high-order bit of the first octet of the stored byte sequence. This is the standard case specified by TIFF *FillOrder* = 1, used in most bitmap interchange and the only case required in Baseline TIFF. This is also the approach used for the octets of standard 8-bit character data, with little attention paid to the fact that the most common forms of data communication transmit and reassemble individual 8-bit frames with the low-order-bit first!

For bit-serial transmission to a distant unit whose approach to assembling bits into bytes is unknown and supposed to be irrelevant, it is necessary to satisfy the expected sequencing of bits over the transmission link. This is the normal case for communication between facsimile units and also for computers and modems emulating standard Group 3 facsimile units. In this case, if the CCITT encoding is captured directly off of the link via standard communication adapters, TIFF *FillOrder* = 2 will usually apply to that stored data form.

Consequently, different TIFF *FillOrder* cases may arise when CCITT encodings are obtained by synthesis within a computer (including Group 4 transmission, which is treated more like computer data) instead of by capture from a Group 3 facsimile unit.

Because this is such a subtle situation, with surprisingly disruptive consequences for *FillOrder* mismatches, the following practice is urged whenever CCITT bilevel encodings are used:

- a. TIFF FillOrder (tag 266) should always be explicitly specified.
- b. FillOrder = 1 should be employed wherever possible in persistent material that is intended for interchange. This is the only reliable case for widespread interchange among computer systems, and it is important to explicitly confirm the honoring of standard assumptions.
- c. FillOrder = 2 should occur only in highly-localized and preferably-transient material, as in a facsimile server supporting group 3 facsimile equipment. The tag should be present as a safeguard against the CCITT encoding “leaking” into an unsuspecting application, allowing readers to detect and warn against the occurrence.

There are interchange situations where fill order is not distinguished, as when filtering the CCITT encoding into a PostScript level 2 image operation. In this case, as in most other cases of computer-based information interchange, FillOrder=1 is assumed, and any padding to a multiple of 8 bits is accomplished by adding a sufficient number of 0-bits to the end of the sequence.

Strips and Tiles. When CCITT bi-level encoding is employed, interaction with stripping (Section 3) and tiling (Section 15) is as follows:

- a. Decompose the image into segments—individual pixel arrays representing the desired strip or tile configuration. The CCITT encoding procedures are applied most flexibly if the segments each have a multiple of 4 lines.
- b. Individually encode each segment according to the specified CCITT bi-level encoding, as if each segment is a separate raster-graphic image.

The reason for this general rule is that CCITT bi-level encodings are generally progressive. That is, the initial line of pixels is encoded, and then subsequent lines, according to a variety of options, are encoded in terms of changes that need to be made to the preceding (unencoded) line. For strips and tiles to be individually usable, they must each start as fresh, independent encodings.

Miscellaneous features. There are provisions in CCITT encoding that are mostly meaningful during facsimile-transmission procedures. There is generally no significant application when storing images in TIFF or other data interchange formats, although TIFF applications should be tolerant and flexible in this regard. These features tend to have significance only when facilitating transfer between facsimile and non-facsimile applications of the encoded raster-graphic images. Further considerations for fill sequences, end-of-line flags, return-to-control (end-of-block) sequences and byte padding are introduced in discussion of the individual encoding options.

T4Options

Tag = 292 (124.H)

Type = LONG

N = 1

See Compression=3. This field is made up of a set of 32 flag bits. Unused bits must be set to 0. Bit 0 is the low-order bit.

Bit 0 is 1 for 2-dimensional coding (otherwise 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 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 1 byte preceded by a zero nibble: xxxx-0000 0000-0001.

Default is 0, for basic 1-dimensional coding. See also Compression.

T6Options

Tag = 293 (125.H)

Type = LONG

N = 1

See *Compression = 4*. This field is made up of a set of 32 flag bits. Unused bits must be set to 0. Bit 0 is the low-order bit. The default value is 0 (all bits 0).

bit 0 is unused and always 0.

bit 1 is 1 if uncompressed mode is allowed in the encoding.

In earlier versions of TIFF, this tag was named Group4Options. The significance has not changed and the present definition is compatible. The name of the tag has been changed to be consistent with the nomenclature of other T.6-encoding applications.

Readers should honor this option tag, and only this option tag, whenever T.6-Encoding is specified for Compression.

For T.6-Encoding, each segment (strip or tile) is encoded as if it were a separate image. The encoded string from each segment starts a fresh byte.

There are no one-dimensional line encodings in T.6-Encoding. Instead, even the first row of the segment’s pixel array is encoded two-dimensionally by always assuming an invisible preceding row of all-white pixels. The 2-dimensional procedure for encoding the body of individual rows is the same as that used for 2-dimensional T.4-encoding and is described fully in the CCITT specifications.

The beginning of the encoding for each row of a strip or tile is conducted as if there is an imaginary preceding (0-width) white pixel, that is as if a fresh run of white pixels has just commenced. The completion of each line is encoded as if there are imaginary pixels beyond the end of the current line, and of the preceding line, in effect, of colors chosen such that the line is exactly completable by a code word, making the imaginary next pixel a changing element that’s not actually used.

The encodings of successive lines follow contiguously in the binary T.6-Encoding stream with no special initiation or separation codewords. There are no provisions for fill codes or explicit end-of-line indicators. The encoding of the last line of the pixel array is followed immediately, in place of any additional line encodings, by a 24-bit End-of-Facsimile Block (EOFB).

000000000001000000000001.B.

The EOFB sequence is immediately followed by enough 0-bit padding to fit the entire stream into a sequence of 8-bit bytes.

General Application. Because of the single uniform encoding procedure, without disruptions by end-of-line codes and shifts into one-dimensional encodings, T.6-encoding is very popular for compression of bi-level images in document imaging systems. T.6-encoding trades off redundancy for minimum encoded size, relying on the underlying storage and transmission systems for reliable retention and communication of the encoded stream.

TIFF readers will operate most smoothly by always ignoring bits beyond the EOFB. Some writers may produce additional bytes of pad bits beyond the byte containing the final bit of the EOFB. Robust readers will not be disturbed by this prospect.

It is not possible to correctly decode a T.6-Encoding without knowledge of the exact number of pixels in each line of the pixel array. ImageWidth (or TileWidth, if used) must be stated exactly and accurately. If an image or segment is overscanned, producing extraneous pixels at the beginning or ending of lines, these pixels must be counted. Any cropping must be accomplished by other means. It is not possible to recover from a pixel-count deviation, even when one is detected. Failure of any row to be completed as expected is cause for abandoning further decoding of the entire segment. There is no requirement that ImageWidth be a multiple of eight, of course, and readers must be prepared to pad the final octet bytes of decoded bitmap rows with additional bits.

If a TIFF reader encounters EOFB before the expected number of lines has been extracted, it is appropriate to assume that the missing rows consist entirely of white pixels. Cautious readers might produce an unobtrusive warning if such an EOFB is followed by anything other than pad bits.

Readers that successfully decode the RowsPerStrip (or TileLength or residual ImageLength) number of lines are not required to verify that an EOFB follows. That is, it is generally appropriate to stop decoding when the expected lines are decoded or the EOFB is detected, whichever occurs first. Whether error indications or warnings are also appropriate depends upon the application and whether more precise troubleshooting of encoding deviations is important.

TIFF writers should always encode the full, prescribed number of rows, with a proper EOFB immediately following in the encoding. Padding should be by the least number of 0-bits needed for the T.6-encoding to exactly occupy a multiple of 8 bits. Only 0-bits should be used for padding, and StripByteCount (or TileByteCount) should not extend to any bytes not containing properly-formed T.6-encoding. In addition, even though not required by T.6-encoding rules, successful interchange with a large variety of readers and applications will be enhanced if writers can arrange for the number of pixels per line and the number of lines per strip to be multiples of eight.

Uncompressed Mode. Although T.6-encodings of simple bi-level images result in data compressions of 10:1 and better, some pixel-array patterns have T.6-encodings that require more bits than their simple bi-level bitmaps. When such cases are detected by encoding procedures, there is an optional extension for shifting to a form of uncompressed coding within the T.6-encoding string.

Uncompressed mode is not well-specified and many applications discourage its usage, preferring alternatives such as different compressions on a segment-by-segment (strip or tile) basis, or by simply leaving the image uncompressed in its

entirety. The main complication for readers is in properly restoring T.6-encoding after the uncompressed sequence is laid down in the current row.

Readers that have no provision for uncompressed mode will generally reject any case in which the flag is set. Readers that are able to process uncompressed-mode content within T.6-encoding strings can safely ignore this flag and simply process any uncompressed-mode occurrences correctly.

Writers that are unable to guarantee the absence of uncompressed-mode material in any of the T.6-encoded segments must set the flag. The flag should be cleared (or defaulted) only when absence of uncompressed-mode material is assured.

Writers that are able to inhibit the generation of uncompressed-mode extensions are encouraged to do so in order to maximize the acceptability of their T.6-encoding strings in interchange situations.

Because uncompressed-mode is not commonly used, the following description is best taken as suggestive of the general machinery. Interpolation of fine details can easily vary between implementations.

Uncompressed mode is signalled by the occurrence of the 10-bit extension code string

```
0000001111.B
```

outside of any run-length make-up code or extension. Original unencoded image information follows. In this unencoded information, a 0-bit evidently signifies a white pixel, a 1-bit signifies a black pixel, and the TIFF PhotometricInterpretation will influence how these bits are mapped into any final uncompressed bitmap for use. The only modification made to the unencoded information is insertion of a 1-bit after every block of five consecutive 0-bits from the original image information. This is a transparency device that allows longer sequences of 0-bits to be reserved for control conditions, especially ending the uncompressed-mode sequence. When it is time to return to compressed mode, the 8-bit exit sequence

```
0000001t.B
```

is appended to the material. The 0-bits of the exit sequence are not considered in applying the 1-bit insertion rule; up to four information 0-bits can legally precede the exit sequence. The trailing bit, 't,' specifies the color (via 0 or 1) that is understood in the next run of compressed-mode encoding. This lets a color other than white be assumed for the 0-width pixel on the left of the edge between the last uncompressed pixel and the resumed 2-dimensional scan.

Writers should confine uncompressed-mode sequences to the interiors of individual rows, never attempting to “wrap” from one row to the next. Readers must operate properly when the only encoding for a single row consists of an uncompressed-mode escape, a complete row of (proper 1-inserted) uncompressed information, and the extension exit. Technically, the exit pixel, 't,' should probably then be the opposite color of the last true pixel of the row, but readers should be generous in this case.

In handling these complex encodings, the encounter of material from a defective source or a corrupted file is particularly unsettling and mysterious. Robust readers will do well to defend against falling off the end of the world; e.g., unexpected EOFB sequences should be handled, and attempted access to data bytes that are not within the bounds of the present segment (or the TIFF file itself) should be avoided.

Section 12: Document Storage and Retrieval

These fields may be useful for document storage and retrieval applications. They will very likely be ignored by other applications.

DocumentName

The name of the document from which this image was scanned.

Tag = 269 (10D.H)

Type = ASCII

See also PageName.

PageName

The name of the page from which this image was scanned.

Tag = 285 (11D.H)

Type = ASCII

See also DocumentName.

No default.

PageNumber

The page number of the page from which this image was scanned.

Tag = 297 (129.H)

Type = SHORT

N = 2

This field is used to specify page numbers of a multiple page (e.g. facsimile) document. PageNumber[0] is the page number; PageNumber[1] is the total number of pages in the document. If PageNumber[1] is 0, the total number of pages in the document is not available.

Pages need not appear in numerical order.

The first page is numbered 0 (zero).

No default.

XPosition

X position of the image.

Tag = 286 (11E.H)

Type = RATIONAL

N = 1

The X offset in ResolutionUnits of the left side of the image, with respect to the left side of the page.

No default. See also YPosition.

YPosition

Y position of the image.

Tag = 287 (11F.H)

Type = RATIONAL

N = 1

The Y offset in ResolutionUnits of the top of the image, with respect to the top of the page. In the TIFF coordinate scheme, the positive Y direction is down, so that YPosition is always positive.

No default. See also XPosition.

Section 13: LZW Compression

This section describes TIFF compression scheme 5, an adaptive compression scheme for raster images.

Restrictions

When LZW compression was added to the TIFF specification, in Revision 5.0, it was thought to be public domain. This is, apparently, not the case.

The following paragraph has been approved by the Unisys Corporation:

“The LZW compression method is said to be the subject of United States patent number 4,558,302 and corresponding foreign patents owned by the Unisys Corporation. Software and hardware developers may be required to license this patent in order to develop and market products using the TIFF LZW compression option. Unisys has agreed that developers may obtain such a license on reasonable, non-discriminatory terms and conditions. Further information can be obtained from: Welch Licensing Department, Office of the General Counsel, M/S C1SW19, Unisys Corporation, Blue Bell, Pennsylvania, 19424.”

Reportedly, there are also other companies with patents that may affect LZW implementors.

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 in very general terms. The author’s goal 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. This section gives sufficient information so that the article is not required reading.

Characteristics

LZW compression has the following characteristics:

- LZW works for images of various bit depths.
- LZW has a reasonable worst-case behavior.
- LZW handles a wide variety of repetitive patterns well.
- LZW is reasonably fast for both compression and decompression.
- LZW does not require floating point software or hardware.

- LZW is lossless. 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.
- LZW works quite well on bilevel images, too. On our test images, it almost always beat PackBits and generally tied CCITT 1D (Modified Huffman) compression. LZW also handles halftoned data better than most bilevel compression schemes.

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 are 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 does not need to be retained for the decompressor. The trick is to make the decompressor automatically build the same table as is built when the data is compressed. 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 {
        = +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 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 in which the LZW character depth equals BitsPerSample, as described in 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 and 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 that handles 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 `ClearCode`, which is defined to be code #256.

is our “prefix string.”

`GetNextCharacter()` retrieves the next character value from the input stream. This will be a number between 0 and 255 because 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 single-character string, and its associated code value, which, in our application, is identical to the character itself. That is, the 0th entry in our table consists of the string <0>, with a corresponding code value of <0>, the 1st entry in the table consists of the string <1>, with a corresponding code value of <1> and the 255th entry in our table consists of the string <255>, with a corresponding code value of <255>.) So, the first entry that added to our string table will be at position 256, right? Well, not quite, because we reserve code #256 for a special “Clear” code. We also reserve code #257 for a special “EndOfInformation” code that we write out at the end of the strip. So the first multiple-character entry added to the string table will be at position 258.

For example, suppose we have input data that looks like this:

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>, because `K` is an empty string at this point. Is the string <7> already in the string table? Of course, because all single character strings were put in the table by `InitializeStringTable()`. So set `K` equal to <7>, and then go to the top of the loop.

Read Pixel 1 into K. Does K (<7><7>) exist in the string table? No, so 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 add K to 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 we write the code associated with (<258>) to output and then 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 we write the code associated with (<8>) to output and then 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. In our example, the output code stream, <7><258><8><8><258><6> 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. *After adding table entry 511, switch to 10-bit codes (i.e., entry 512 should be a 10-bit code.) Likewise, switch to 11-bit codes after table entry 1023, and 12-bit codes after table entry 2047.* We will arbitrarily limit ourselves to 12-bit codes, so that our table can have at most 4096 entries. The table should not be any larger.

Whenever you add a code to the output stream, it “counts” toward the decision about bumping the code bit length. This is important when writing the last code word before an EOI code or ClearCode, to avoid code length errors.

What happens if we run out of room in our string table? This is where the ClearCode comes in. As soon as we use entry 4094, we write out a (12-bit) ClearCode. (If we wait any longer to write the ClearCode, the decompressor might try to interpret the ClearCode as a 13-bit code.) At this point, the compressor reinitializes the string table and then writes 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 ClearCode. You can either write it out as a 12-bit code before writing the ClearCode, in which case you need to do it right after adding table entry 4093, or

you can write it as a 9-bit code after the ClearCode . 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 ClearCode 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 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. Thus, it reflects the characteristics of the data, providing a high degree of adaptability.

LZW Decoding

The procedure for decompression is a little more complicated:

```

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
)+FirstChar(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 #510 is stored into the table. *Similarly, the switch is made to 11-bit codes after #1022 and to 12-bit codes after #2046.*

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` indicates string concatenation.

`StringFromCode()` looks up the string associated with a given code.

`WriteString()` adds a string to the output stream.

When SamplesPerPixel Is Greater Than 1

So far, we have described the compression scheme as if `SamplesPerPixel` were always 1, as is 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 whether `PlanarConfiguration=1` or `PlanarConfiguration=2`, for RGB images. So, use whichever configuration you prefer and simply compress the bytes in the strip.

Note: Compression ratios on our test RGB images were disappointingly low: 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 as possible from their images. 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 can be effective, producing 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 more straightforward and faster because no search is involved—strings can be accessed directly by code value.

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. See the following Section.

Acknowledgments

See the first page of this section for the LZW reference.

The use of ClearCode as a technique for handling 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 uses an adaptation of the LZW technique.

Section 14: Differencing Predictor

This section defines a Predictor that greatly improves compression ratios for some images.

Predictor

Tag = 317 (13D.H)

Type = SHORT

N = 1

A predictor is a mathematical operator that is applied to the image data before an encoding scheme is applied. Currently this field is used only with LZW (Compression=5) encoding because LZW is probably the only TIFF encoding scheme that benefits significantly from a predictor step. See Section 13.

The possible values are:

- 1 = No prediction scheme used before coding.
- 2 = Horizontal differencing.

Default is 1.

The algorithm

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 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++)
    for (col = ncols - 1; col >= 1; col--)
        image[row][col] -= image[row][col-1];
```

If we don't have 8-bit components, we need to work a little harder to make better use of the architecture of most CPUs. Suppose we have 4-bit components packed two per byte in the normal TIFF uncompressed (i.e., Compression=1) fashion. To find differences, we want to first expand each 4-bit component into an 8-bit byte, so that we have one component per byte, low-order justified. We then perform the horizontal differencing illustrated in the example above. Once the differencing has been completed, we then repack the 4-bit differences two to a byte, in the normal TIFF uncompressed fashion.

If the components are greater than 8 bits deep, expanding the components 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 seem at first that our differencing might turn 8-bit components into 9-bit differences, 4-bit components 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 two’s 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 works the same as it does for grayscale data.

If `PlanarConfiguration` is 1, however, things get a little trickier. If we didn’t do anything special, we would subtract red component values from green component values, green component values from blue component values, and blue component values from red component values. This 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 components.

Results and Guidelines

LZW without differencing works well for 1-bit images, 4-bit grayscale images, and many palette-color images. But natural 24-bit color images and some 8-bit grayscale images do much better with differencing.

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 components. The simplest way to get rid of noise is to mask off one or two low-order bits of each 8-bit component. 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 component, 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.

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.

Section 15: Tiled Images

Introduction

Motivation

This section describes how to organize images into tiles instead of strips.

For low-resolution to medium-resolution images, the standard TIFF method of breaking the image into strips is adequate. However high-resolution images can be accessed more efficiently—and compression tends to work better—if the image is broken into roughly square tiles instead of horizontally-wide but vertically-narrow strips.

Relationship to existing fields

When the tiling fields described below are used, they replace the StripOffsets, StripByteCounts, and RowsPerStrip fields. Use of tiles will therefore cause older TIFF readers to give up because they will have no way of knowing where the image data is or how it is organized. **Do not** use both strip-oriented and tile-oriented fields in the same TIFF file.

Padding

Tile size is defined by TileWidth and TileLength. All tiles in an image are the same size; that is, they have the same pixel dimensions.

Boundary tiles are padded to the tile boundaries. For example, if TileWidth is 64 and ImageWidth is 129, then the image is 3 tiles wide and 63 pixels of padding must be added to fill the rightmost column of tiles. The same holds for TileLength and ImageLength. It doesn't matter what value is used for padding, because good TIFF readers display only the pixels defined by ImageWidth and ImageLength and ignore any padded pixels. Some compression schemes work best if the padding is accomplished by replicating the last column and last row instead of padding with 0's.

The price for padding the image out to tile boundaries is that some space is wasted. But compression usually shrinks the padded areas to almost nothing. Even if data is not compressed, remember that tiling is intended for large images. Large images have lots of comparatively small tiles, so that the percentage of wasted space will be very small, generally on the order of a few percent or less.

The advantages of padding an image to the tile boundaries are that implementations can be simpler and faster and that it is more compatible with tile-oriented compression schemes such as JPEG. See Section 22.

Tiles are compressed individually, just as strips are compressed. *That is, each row of data in a tile is treated as a separate “scanline” when compressing.* Compress-

sion includes any padded areas of the rightmost and bottom tiles so that all the tiles in an image are the same size when uncompressed.

All of the following fields are required for tiled images:

Fields

TileWidth

Tag = 322 (142.H)

Type = SHORT or LONG

N = 1

The tile width in pixels. This is the number of columns in each tile.

Assuming integer arithmetic, three computed values that are useful in the following field descriptions are:

$$\text{TilesAcross} = (\text{ImageWidth} + \text{TileWidth} - 1) / \text{TileWidth}$$

$$\text{TilesDown} = (\text{ImageLength} + \text{TileLength} - 1) / \text{TileLength}$$

$$\text{TilesPerImage} = \text{TilesAcross} * \text{TilesDown}$$

These computed values are not TIFF fields; they are simply values determined by the ImageWidth, TileWidth, ImageLength, and TileLength fields.

TileWidth and ImageWidth together determine the number of tiles that span the width of the image (TilesAcross). TileLength and ImageLength together determine the number of tiles that span the length of the image (TilesDown).

We recommend choosing TileWidth and TileLength such that the resulting tiles are about 4K to 32K bytes before compression. This seems to be a reasonable value for most applications and compression schemes.

TileWidth must be a multiple of 16. This restriction improves performance in some graphics environments and enhances compatibility with compression schemes such as JPEG.

Tiles need not be square.

Note that ImageWidth can be less than TileWidth, although this means that the tiles are too large or that you are using tiling on really small images, neither of which is recommended. The same observation holds for ImageLength and TileLength.

No default. See also TileLength, TileOffsets, TileByteCounts.

TileLength

Tag = 323 (143.H)

Type = SHORT or LONG

N = 1

The tile length (height) in pixels. This is the number of rows in each tile.

TileLength must be a multiple of 16 for compatibility with compression schemes such as JPEG.

Replaces RowsPerStrip in tiled TIFF files.

No default. See also TileWidth, TileOffsets, TileByteCounts.

TileOffsets

Tag = 324 (144.H)

Type = LONG

N = TilesPerImage for PlanarConfiguration = 1

= SamplesPerPixel * TilesPerImage for PlanarConfiguration = 2

For each tile, the byte offset of that tile, as compressed and stored on disk. The offset is specified with respect to the beginning of the TIFF file. Note that this implies that each tile has a location independent of the locations of other tiles.

Offsets are ordered left-to-right and top-to-bottom. For PlanarConfiguration = 2, the offsets for the first component plane are stored first, followed by all the offsets for the second component plane, and so on.

No default. See also TileWidth, TileLength, TileByteCounts.

TileByteCounts

Tag = 325 (145.H)

Type = SHORT or LONG

N = TilesPerImage for PlanarConfiguration = 1

= SamplesPerPixel * TilesPerImage for PlanarConfiguration = 2

For each tile, the number of (compressed) bytes in that tile.

See TileOffsets for a description of how the byte counts are ordered.

No default. See also TileWidth, TileLength, TileOffsets.

Section 16: CMYK Images

Motivation

This section describes how to store separated (usually CMYK) image data in a TIFF file.

In a separated image, each pixel consists of N components. Each component represents the amount of a particular ink that is to be used to represent the image at that location, typically using a halftoning technique.

For example, in a CMYK image, each pixel consists of 4 components. Each component represents the amount of cyan, magenta, yellow, or black process ink that is to be used to represent the image at that location.

The fields described in this section can be used for more than simple 4-color process (CMYK) printing. They can also be used for describing an image made up of more than 4 inks, such an image made up of a cyan, magenta, yellow, red, green, blue, and black inks. Such an image is sometimes called a high-fidelity image and has the advantage of slightly extending the printed color gamut.

Since separated images are quite device-specific and are restricted to color prepress use, they should not be used for general image data interchange. Separated images are to be used only for prepress applications in which the imagesetter, paper, ink, and printing press characteristics are known by the creator of the separated image.

Note: there is no single method of converting RGB data to CMYK data and back. In a perfect world, something close to cyan = 255-red, magenta = 255-green, and yellow = 255-blue should work; but characteristics of printing inks and printing presses, economics, and the fact that the meaning of RGB itself depends on other parameters combine to spoil this simplicity.

Requirements

In addition to satisfying the normal Baseline TIFF requirements, a separated TIFF file must have the following characteristics:

- **SamplesPerPixel = N .** SHORT. The number of inks. (For example, $N=4$ for CMYK, because we have one component each for cyan, magenta, yellow, and black.)
- **BitsPerSample = 8,8,8,8 (for CMYK).** SHORT. For now, only 8-bit components are recommended. The value “8” is repeated SamplesPerPixel times.
- **PhotometricInterpretation = 5 (Separated - usually CMYK).** SHORT. The components represent the desired percent dot coverage of each ink, where the larger component values represent a higher percentage of ink dot coverage and smaller values represent less coverage.

Fields

In addition, there are some new fields, all of which are optional.

InkSet

Tag = 332 (14C.H)

Type = SHORT

N = 1

The set of inks used in a separated (PhotometricInterpretation=5) image.

1 = CMYK. The order of the components is cyan, magenta, yellow, black. Usually, a value of 0 represents 0% ink coverage and a value of 255 represents 100% ink coverage for that component, but see DotRange below. The InkNames field should not exist when InkSet=1.

2 = not CMYK. See the InkNames field for a description of the inks to be used.

Default is 1 (CMYK).

NumberOfInks

Tag = 334 (14E.H)

Type = SHORT

N = 1

The number of inks. Usually equal to SamplesPerPixel, unless there are extra samples.

See also ExtraSamples.

Default is 4.

InkNames

Tag = 333 (14D.H)

Type = ASCII

N = total number of characters in all the ink name strings, including the NULs.

The name of each ink used in a separated (PhotometricInterpretation=5) image, written as a list of concatenated, NUL-terminated ASCII strings. The number of strings must be equal to NumberOfInks.

The samples are in the same order as the ink names.

See also InkSet, NumberOfInks.

No default.

DotRange

Tag = 336 (150.H)

Type = BYTE or SHORT

N = 2, or 2*SamplesPerPixel

The component values that correspond to a 0% dot and 100% dot. DotRange[0] corresponds to a 0% dot, and DotRange[1] corresponds to a 100% dot.

If a DotRange pair is included for each component, the values for each component are stored together, so that the pair for Cyan would be first, followed by the pair for Magenta, and so on. *Use of multiple dot ranges is, however, strongly discouraged in the interests of simplicity and compatibility with ANSI IT8 standards.*

A number of prepress systems like to keep some “headroom” and “footroom” on both ends of the range. What to do with components that are less than the 0% aim point or greater than the 100% aim point is not specified and is application-dependent.

It is strongly recommended that a CMYK TIFF writer not attempt to use this field to reverse the sense of the pixel values so that smaller values mean more ink instead of less ink. That is, DotRange[0] should be less than DotRange[1].

DotRange[0] and DotRange[1] must be within the range $[0, (2^{**}BitsPerSample) - 1]$.

Default: a component value of 0 corresponds to a 0% dot, and a component value of 255 (assuming 8-bit pixels) corresponds to a 100% dot. That is, DotRange[0] = 0 and DotRange[1] = $(2^{**}BitsPerSample) - 1$.

TargetPrinter

Tag = 337 (151.H)

Type = ASCII

N = any

A description of the printing environment for which this separation is intended.

History

This Section has been expanded from earlier drafts, with the addition of the **InkSet**, **InkNames**, **NumberOfInks**, **DotRange**, and **TargetPrinter**, but is backward-compatible with earlier draft versions.

Possible future enhancements: definition of the characterization information so that the CMYK data can be retargeted to a different printing environment and so that display on a CRT or proofing device can more accurately represent the color. ANSI IT8 is working on such a proposal.

Section 17: HalftoneHints

This section describes a scheme for properly placing highlights and shadows in halftoned images.

Introduction

The single most easily recognized failing of continuous tone images is the incorrect placement of highlight and shadow. It is critical that a halftone process be capable of printing the lightest areas of the image as the smallest halftone spot capable of the output device, at the specified printer resolution and screen ruling. Specular highlights (small ultra-white areas) as well as the shadow areas should be printable as paper only.

Consistency in highlight and shadow placement allows the user to obtain predictable results on a wide variety of halftone output devices. Proper implementation of the `HalftoneHints` field will provide a significant step toward device independent imaging, such that low cost printers may be used as effective proofing devices for images which will later be halftoned on a high-resolution imagesetter.

The HalftoneHints Field

HalftoneHints

Tag = 321 (141.H)

Type = SHORT

N = 2

The purpose of the `HalftoneHints` field is to convey to the halftone function the range of gray levels within a colorimetrically-specified image that should retain tonal detail. The field contains two values of sixteen bits each and, therefore, is contained wholly within the field itself; no offset is required. The first word specifies the highlight gray level which should be halftoned at the lightest printable tint of the final output device. The second word specifies the shadow gray level which should be halftoned at the darkest printable tint of the final output device. Portions of the image which are whiter than the highlight gray level will quickly, if not immediately, fade to specular highlights. There is no default value specified, since the highlight and shadow gray levels are a function of the subject matter of a particular image.

Appropriate values may be derived algorithmically or may be specified by the user, either directly or indirectly.

The `HalftoneHints` field, as defined here, defines an achromatic function. It can be used just as effectively with color images as with monochrome images. When used with opponent color spaces such as CIE $L^*a^*b^*$ or YCbCr, it refers to the achromatic component only; L^* in the case of CIELab, and Y in the case of

YCbCr. When used with tri-stimulus spaces such as RGB, it suggests to retain tonal detail for all colors with an NTSC gray component within the bounds of the R=G=B=Highlight to R=G=B=Shadow range.

Comments for TIFF Writers

TIFF writers are encouraged to include the HalftoneHints field in all color or grayscale images where BitsPerSample >1. Although no default value is specified, prior to the introduction of this field it has been common practice to implicitly specify the highlight and shadow gray levels as 1 and $2^{**}BitsperSample-2$ and manipulate the image data to this definition. There are some disadvantages to this technique, and it is not feasible for a fixed gamut colorimetric image type. Appropriate values may be derived algorithmically or may be specified by the user directly or indirectly. Automatic algorithms exist for analyzing the histogram of the achromatic intensity of an image and defining the minimum and maximum values as the highlight and shadow settings such that tonal detail is retained throughout the image. This kind of algorithm may try to impose a highlight or shadow where none really exists in the image, which may require user controls to override the automatic setting.

It should be noted that the choice of the highlight and shadow values is somewhat output dependent. For instance, in situations where the dynamic range of the output medium is very limited (as in newsprint and, to a lesser degree, laser output), it may be desirable for the user to clip some of the lightest or darkest tones to avoid the reduced contrast resulting from compressing the tone of the entire image. Different settings might be chosen for 150-line halftone printed on coated stock. Keep in mind that these values may be adjusted later (which might not be possible unless the image is stored as a colorimetric, fixed, full-gamut image), and that more sophisticated page-layout applications may be capable of presenting a user interface to consider these decisions at a point where the halftone process is well understood.

It should be noted that although CCDs are linear intensity detectors, TIFF writers may choose to manipulate the image to store gamma-compensated data. Gamma-compensated data is more efficient at encoding an image than is linear intensity data because it requires fewer BitsPerPixel to eliminate banding in the darker tones. It also has the advantage of being closer to the tone response of the display or printer and is, therefore, less likely to produce poor results from applications that are not rigorous about their treatment of images. Be aware that the PhotometricInterpretation value of 0 or 1 (grayscale) implies linear data because no gamma is specified. The PhotometricInterpretation value of 2 (RGB data) specifies the NTSC gamma of 2.2 as a default. If data is written as something other than the default, then a GrayResponseCurve field or a TransferFunction field must be present to define the deviation. For grayscale data, be sure that the densities in the GrayResponseCurve are consistent with the PhotometricInterpretation field and the HalftoneHints field.

Comments for TIFF Readers

TIFF readers that send a grayscale image to a halftone output device, whether it is a binary laser printer or a PostScript imagesetter should make an effort to maintain the highlight and shadow placement. This requires two steps. First, determine the highlight and shadow gray level of a particular image. Second, communicate that information to the halftone engine.

To determine the highlight and shadow gray levels, begin by looking for a `HalftoneHints` field. If it exists, it takes precedence. The first word represents the gray level of the highlight and the second word represents the gray level of the shadow. If the image is a colorimetric image (i.e. it has a `GrayResponseCurve` field or a `TransferFunction` field) but does not contain a `HalftoneHints` field, then the gamut mapping techniques described earlier should be used to determine the highlight and shadow values. If neither of these conditions are true, then the file should be treated as if a `HalftoneHints` field had indicated a highlight at gray level 1 and a shadow at gray level $2^{**}BitsPerPixel-2$ (or vice-versa depending on the `PhotometricInterpretation` field). Once the highlight and shadow gray levels have been determined, the next step is to communicate this information to the halftone module. The halftone module may exist within the same application as the TIFF reader, it may exist within a separate printer driver, or it may exist within the Raster Image Processor (RIP) of the printer itself. Whether the halftone process is a simple dither pattern or a general purpose spot function, it has some gray level at which the lightest printable tint will be rendered. The `HalftoneHint` concept is best implemented in an environment where this lightest printable tint is easily and consistently specified.

There are several ways in which an application can communicate the highlight and shadow to the halftone function. Some environments may allow the application to pass the highlight and shadow to the halftone module explicitly along with the image. This is the best approach, but many environments do not yet provide this capability. Other environments may provide fixed gray levels at which the highlight and shadow will be rendered. For these cases, the application should build a tone map that matches the highlight and shadow specified in the image to the highlight and shadow gray level of the halftone module. This approach requires more work by the application software, but will provide excellent results. Some environments will not have any consistent concept of highlight and shadow at all. In these environments, the best an application can do is characterize each of the supported printers and save the observed highlight and shadow gray levels. The application can then use these values to achieve the desired results, providing the environment doesn't change.

Once the highlight and shadow areas are selected, care should be taken to appropriately map intermediate gray levels to those expected by the halftone engine, which may or may not be linear Reflectance. Note that although CCDs are linear intensity detectors and many TIFF files are stored as linear intensity, most output devices require significant tone compensation (sometimes called gamma correction) to correctly display or print linear data. Be aware that the `PhotometricInterpretation` value of 0, 1 implies linear data because no gamma is specified. The `PhotometricInterpretation` value of 2 (RGB data) specifies the NTSC gamma of 2.2 as a default. If a `GrayResponseCurve` field or a `TransferFunction` field is present, it may define something other than the default.

Some Background on the Halftone Process

To obtain the best results when printing a continuous-tone raster image, it is seldom desirable to simply reproduce the tones of the original on the printed page. Most often there is some gamut mapping required. Often this is because the tonal range of the original extends beyond the tonal range of the output medium. In some cases, the tone range of the original is within the gamut of the output medium, but it may be more pleasing to expand the tone of the image to fill the range of the output. Given that the tone of the original is to be adjusted, there is a whole range of possibilities for the level of sophistication that may be undertaken by a software application.

Printing monochrome output is far less sophisticated than printing color output. For monochrome output the first priority is to control the placement of the highlight and the shadow. Ideally, a quality halftone will have sufficient levels of gray so that a standard observer cannot distinguish the interface between any two adjacent levels of gray. In practice, however, there is often a significant step between the tone of the paper and the tone of the lightest printable tint. Although usually less severe, the problem is similar between solid ink and the darkest printable tint. Since the dynamic range between the lightest printable tint and the darkest printable tint is usually less than one would like, it is common to maximize the tone of the image within these bounds. Not all images will have a highlight (an area of the image which is desirable to print as light as possible while still retaining tonal detail). If one exists, it should be carefully controlled to print at the lightest printable tint of the output medium. Similarly, the darkest areas of the image to retain tonal detail should be printed as the darkest printable tint of the output medium. Tones lighter or darker than these may be clipped at the limits of the paper and ink. Satisfactory results may be obtained in monochrome work by doing nothing more than a perceptually-linear mapping of the image between these rigorously controlled endpoints. This level of sophistication is sufficient for many mid-range applications, although the results often appear flatter (i.e. lower in contrast) than desired.

The next step is to increase contrast slightly in the tonal range of the image that contains the most important subject matter. To perform this step well requires considerably more information about the image and about the press. To know where to add contrast, the algorithm must have access to first the keyness of the image; the tone range which the user considers most important. To know how much contrast to add, the algorithm must have access to the absolute tone of the original and the dynamic range of the output device so that it may calculate the amount of tone compression to which the image is actually subjected.

Most images are called normal key. The important subject areas of a normal key image are in the midtones. These images do well when a so-called “sympathetic curve” is applied, which increases the contrast in midtones slightly at the expense of contrast in the lighter and darker tones. White china on a white tablecloth is an example of a high key image. High key images benefit from higher contrast in lighter tones, with less contrast needed in the midtones and darker tones. Low key images have important subject matter in the darker tones and benefit from increasing the contrast in the darker tones. Specifying the keyness of an image might be attempted by automatic techniques, but it will likely fail without user input. For example, a photo of a bride in a white wedding dress it may be a high key image if

you are selling wedding dresses, but may be a normal key image if you are the parents of the bride and are more interested in her smile.

Sophisticated color reproduction employs all of these principles, and then applies them in three dimensions. The mapping of the highlight and shadow becomes only one small, albeit critical, portion of the total issue of mapping colors that are too saturated for the output medium. Here again, automatic techniques may be employed as a first pass, with the user becoming involved in the clip or compress mapping decision. The HalftoneHints field is still useful in communicating which portions of the intensity of the image must be retained and which may be clipped. Again, a sophisticated application may override these settings if later user input is received.

Section 18: Associated Alpha Handling

This section describes a scheme for handling images with alpha data.

Introduction

A common technique in computer graphics is to assemble an image from one or more elements that are rendered separately. When elements are combined using compositing techniques, matte or coverage information must be present for each pixel to create a properly anti-aliased accumulation of the full image [Porter84]. This matting information is an example of additional per-pixel data that must be maintained with an image. This section describes how to use the ExtraSamples field to store the requisite matting information, commonly called the associated alpha or just alpha. This scheme enables efficient manipulation of image data during compositing operations.

Images with matting information are stored in their natural format but with an additional component per pixel. The ExtraSample field is included with the image to indicate that an extra component of each pixel contains associated alpha data. In addition, when associated alpha data are included with RGB data, the RGB components must be stored premultiplied by the associated alpha component and component values in the range $[0, 2^{BitsPerSample}-1]$ are implicitly mapped onto the $[0, 1]$ interval. That is, for each pixel (r, g, b) and opacity A , where r , g , b , and A are in the range $[0, 1]$, $(A*r, A*g, A*b, A)$ must be stored in the file. If A is zero, then the color components should be interpreted as zero. Storing data in this pre-multiplied format, allows compositing operations to be implemented most efficiently. In addition, storing pre-multiplied data makes it possible to specify colors with components outside the normal $[0, 1]$ interval. The latter is useful for defining certain operations that effect only the luminescence [Porter84].

Fields

ExtraSamples

Tag = 338 (152.H)

Type = SHORT

N = 1

This field must have a value of 1 (associated alpha data with pre-multiplied color components). The associated alpha data stored in component SamplesPerPixel-1 of each pixel contains the opacity of that pixel, and the color information is pre-multiplied by alpha.

Comments

Associated alpha data is just another component added to each pixel. Thus, for example, its size is defined by the value of the BitsPerSample field.

Note that since data is stored with RGB components already multiplied by alpha, naive applications that want to display an RGBA image on a display can do so simply by displaying the RGB component values. This works because it is effectively the same as merging the image with a black background. That is, to merge one image with another, the color of resultant pixels are calculated as:

$$C_r = C_{over} * A_{over} + C_{under} * (I - A_{over})$$

Since the “under image” is a black background, this equation reduces to

$$C_r = C_{over} * A_{over}$$

which is exactly the pre-multiplied color; i.e. what is stored in the image.

On the other hand, to print an RGBA image, one must composite the image over a suitable background page color. For a white background, this is easily done by adding 1 - A to each color component. For an arbitrary background color C_{back} , the *printed color* of each pixel is

$$C_{print} = C_{image} + C_{back} * (I - A_{image})$$

(since C_{image} is pre-multiplied).

Since the ExtraSamples field is independent of other fields, this scheme permits alpha information to be stored in whatever organization is appropriate. In particular, components can be stored packed (PlanarConfiguration=1); this is important for good I/O performance and for good memory access performance on machines that are sensitive to data locality. However, if this scheme is used, TIFF readers must not derive the SamplesPerPixel from the value of the PhotometricInterpretation field (e.g., if RGB, then SamplesPerPixel is 3).

In addition to being independent of data storage-related fields, the field is also independent of the PhotometricInterpretation field. This means, for example, that it is easy to use this field to specify grayscale data and associated matte information. Note that a palette-color image with associated alpha will not have the colormap indices pre-multiplied; rather, the RGB colormap values will be pre-multiplied.

Unassociated Alpha and Transparency Masks

Some image manipulation applications support notions of transparency masks and soft-edge masks. The associated alpha information described in this section is different from this *unassociated alpha* information in many ways, most importantly:

- Associated alpha describes opacity or coverage at each pixel, while clipping-related alpha information describes a boolean relationship. That is, associated alpha can specify fractional coverage at a pixel, while masks specify either 0 or 100 percent coverage.
- Once defined, associated alpha is not intended to be removed or edited, except as a result of compositing the image; it is an integral part of an image.

Unassociated alpha, on the other hand, is designed as an ancillary piece of information.

References

[Porter84] “Compositing Digital Images”. Thomas Porter, Tom Duff; Lucasfilm Ltd. ACM SIGGRAPH Proceedings Volume 18, Number 3. July, 1984.

Section 19: Data Sample Format

This section describes a scheme for specifying data sample type information.

TIFF implicitly types all data samples as unsigned integer values. Certain applications, however, require the ability to store image-related data in other formats such as floating point. This section presents a scheme for describing a variety of data sample formats.

Fields

SampleFormat

Tag = 339 (153.H)

Type = SHORT

N = SamplesPerPixel

This field specifies how to interpret each data sample in a pixel. Possible values are:

- 1 = unsigned integer data
- 2 = two's complement signed integer data
- 3 = IEEE floating point data [IEEE]
- 4 = undefined data format

Note that the *SampleFormat* field does not specify the size of data samples; this is still done by the *BitsPerSample* field.

A field value of “undefined” is a statement by the writer that it did not know how to interpret the data samples; for example, if it were copying an existing image. A reader would typically treat an image with “undefined” data as if the field were not present (i.e. as unsigned integer data).

Default is 1, unsigned integer data.

SMinSampleValue

Tag = 340 (154.H)

Type = the field type that best matches the sample data

N = SamplesPerPixel

This field specifies the minimum sample value. Note that a value should be given for each data sample. That is, if the image has 3 *SamplesPerPixel*, 3 values must be specified.

The default for *SMinSampleValue* and *SMaxSampleValue* is the full range of the data type.

SMaxSampleValue

Tag = 341 (155.H)

Type = the field type that best matches the sample data

N = SamplesPerPixel

This new field specifies the maximum sample value.

Comments

The SampleFormat field allows more general imaging (such as image processing) applications to employ TIFF as a valid file format.

SMinSampleValue and SMaxSampleValue become more meaningful when image data is typed. The presence of these fields makes it possible for readers to assume that data samples are bound to the range [SMinSampleValue, SMaxSampleValue] without scanning the image data.

References

[IEEE] “IEEE Standard 754 for Binary Floating-point Arithmetic”.

Section 20: RGB Image Colorimetry

Without additional information, RGB data is device-specific; that is, without an absolute color meaning. This section describes a scheme for describing and characterizing RGB image data.

Introduction

Color printers, displays, and scanners continue to improve in quality and availability while they drop in price. Now the problem is to display color images so that they appear to be identical on different hardware.

The key to reproducing the same color on different devices is to use the CIE 1931 XYZ color-matching functions, the international standard for color comparison. Using CIE XYZ, an image's colorimetry information can fully describe its color interpretation. The approach taken here is essentially calibrated RGB. It implies a transformation from the RGB color space of the pixels to CIE 1931 XYZ.

The appearance of a color depends not only on its absolute tristimulus values, but also on the conditions under which it is viewed, including the nature of the surround and the adaptation state of the viewer. Colors having the same absolute tristimulus values appear the same in identical viewing conditions. The more complex issue of color appearance under different viewing conditions is addressed by [4]. The colorimetry information presented here plays an important role in color appearance under different viewing conditions.

Assuming identical viewing conditions, an application using the tags described below can display an image on different hardware and achieve colorimetrically identical results. The process of using this colorimetry information for displaying an image is straightforward on a color monitor but it is more complex for color printers. Also, the results will be limited by the color gamut and other characteristics of the display or printing device.

The following fields describe the image colorimetry information of a TIFF image:

WhitePoint chromaticity of the white point of the image

PrimaryChromaticities chromaticities of the primaries of the image

TransferFunction transfer function for the pixel data

TransferRange extends the range of the transfer function

ReferenceBlackWhite pixel component headroom and footroom parameters

The *TransferFunction*, *TransferRange*, and *ReferenceBlackWhite* fields have defaults based on industry standards. An image has a colorimetric interpretation if and only if both the *WhitePoint* and *PrimaryChromaticities* fields are present. An image without these colorimetry fields will be displayed in an application and hardware dependent manner.

Note: In the following definitions, *BitsPerSample* is used as if it were a single number when in fact it is an array of *SamplesPerPixel* numbers. The elements of

this array may not always be equal, for example: 5/6/5 16-bit pixels. BitsPerSample should be interpreted as the BitsPerSample value associated with a particular component. In the case of unequal BitsPerSample values, the definitions below can be extended in a straightforward manner.

This section has the following differences with Appendix H in TIFF 5.0:

- removed the use of image colorimetry defaults
- renamed the ColorResponseCurves field as TransferFunction
- optionally allowed a single TransferFunction table to describe all three channels
- described the use of the TransferFunction field for YCbCr, Palette, WhiteIsZero and BlackIsZero PhotometricInterpretation types
- added the TransferRange tag to expand the range of the TransferFunction below black and above white
- added the ReferenceBlackWhite field
- addressed the issue of color appearance

Colorimetry Field Definitions

WhitePoint

Tag = 318 (13E.H)

Type = RATIONAL

N = 2

The chromaticity of the white point of the image. This is the chromaticity when each of the primaries has its ReferenceWhite value. The value is described using the 1931 CIE xy chromaticity diagram and only the chromaticity is specified. This value can correspond to the chromaticity of the alignment white of a monitor, the filter set and light source combination of a scanner or the imaging model of a rendering package. The ordering is white[x], white[y].

For example, the CIE Standard Illuminant D65 used by CCIR Recommendation 709 and Kodak PhotoYCC is:

3127/10000,3290/10000

No default.

PrimaryChromaticities

Tag =319 (13F.H)

Type = RATIONAL

N = 6

The chromaticities of the primaries of the image. This is the chromaticity for each of the primaries when it has its ReferenceWhite value and the other primaries have their ReferenceBlack values. These values are described using the 1931 CIE xy chromaticity diagram and only the chromaticities are specified. These values can correspond to the chromaticities of the phosphors of a monitor, the filter set and light source combination of a scanner or the imaging model of a rendering package. The ordering is red[x], red[y], green[x], green[y], blue[x], and blue[y].

For example the CCIR Recommendation 709 primaries are:

640/1000,330/1000,

300/1000, 600/1000,

150/1000, 60/1000

No default.

TransferFunction

Tag =301 (12D.H)

Type = SHORT

N = {1 or 3} * (1 << BitsPerSample)

Describes a transfer function for the image in tabular style. Pixel components can be gamma-compensated, companded, non-uniformly quantized, or coded in some other way. The TransferFunction maps the pixel components from a non-linear BitsPerSample (e.g. 8-bit) form into a 16-bit linear form without a perceptible loss of accuracy.

If $N = 1 \ll \text{BitsPerSample}$, the transfer function is the same for each channel and all channels share a single table. Of course, this assumes that each channel has the same BitsPerSample value.

If $N = 3 * (1 \ll \text{BitsPerSample})$, there are three tables, and the ordering is the same as it is for pixel components of the PhotometricInterpretation field. These tables are separate and not interleaved. For example, with RGB images all red entries come first, followed by all green entries, followed by all blue entries.

The length of each component table is $1 \ll \text{BitsPerSample}$. The width of each entry is 16 bits as implied by the type SHORT. Normally the value 0 represents the minimum intensity and 65535 represents the maximum intensity and the values [0, 0, 0] represent black and [65535,65535, 65535] represent white. If the TransferRange tag is present then it is used to determine the minimum and maximum values, and a scaling normalization.

The TransferFunction can be applied to images with a PhotometricInterpretation value of RGB, Palette, YCbCr, WhiteIsZero, and BlackIsZero. The TransferFunction is not used with other PhotometricInterpretation types.

For RGB PhotometricInterpretation, ReferenceBlackWhite expands the coding range, TransferRange expands the range of the TransferFunction, and the TransferFunction tables decompand the RGB value. The WhitePoint and PrimaryChromaticities further describe the RGB colorimetry.

For Palette color PhotometricInterpretation, the Colormap maps the pixel into three 16-bit values that when scaled to BitsPerSample-bits serve as indices into the TransferFunction tables which decompand the RGB value. The WhitePoint and PrimaryChromaticities further describe the underlying RGB colorimetry.

A Palette value can be scaled into a TransferFunction index by:

$$\text{index} = (\text{value} * ((1 \ll \text{BitsPerSample}) - 1)) / 65535;$$

A TransferFunction index can be scaled into a Palette color value by:

$$\text{value} = (\text{index} * 65535L) / ((1 \ll \text{BitsPerSample}) - 1);$$

Be careful if you intend to create Palette images with a TransferFunction. If the Colormap tag is directly converted from a hardware colormap, it may have a device gamma already incorporated into the DAC values.

For YCbCr PhotometricInterpretation, ReferenceBlackWhite expands the coding range, the YCbCrCoefficients describe the decoding matrix to transform YCbCr into RGB, TransferRange expands the range of the TransferFunction, and the TransferFunction tables decompand the RGB value. The WhitePoint and PrimaryChromaticities fields provide further description of the underlying RGB colorimetry.

After coding range expansion by ReferenceBlackWhite and TransferFunction expansion by TransferRange, RGB values may be outside the domain of the TransferFunction. Also, the display device matrix can transform RGB values into display device RGB values outside the domain of the device. These values are handled in an application-dependent manner.

For RGB images with non-default ReferenceBlackWhite coding range expansion and for YCbCr images, the resolution of the TransferFunction may be insufficient. For example, after the YCbCr transformation matrix, the decoded RGB values must be rounded to index into the TransferFunction tables. Applications needing the extra accuracy should interpolate between the elements of the TransferFunction tables. Linear interpolation is recommended.

For WhiteIsZero and BlackIsZero PhotometricInterpretation, the TransferFunction decompands the grayscale pixel value to a linear 16-bit form. Note that a TransferFunction value of 0 represents black and 65535 represents white regardless of whether a grayscale image is WhiteIsZero or BlackIsZero. For example, the zeroth element of a WhiteIsZero TransferFunction table will likely be 65535. This extension of the TransferFunction field for grayscale images is intended to replace the GrayResponseCurve field.

The TransferFunction does not describe a transfer characteristic outside of the range for ReferenceBlackWhite.

Default is a single table corresponding to the NTSC standard gamma value of 2.2. This table is used for each channel. It can be generated by:

```
NValues = 1 << BitsPerSample;
for (TF[0]= 0, i = 1; i < NValues; i++)
    TF[i]= floor(pow(i / (NValues - 1.0), 2.2) * 65535 + 0.5);
```

TransferRange

Tag = 342 (156.H)

Type = SHORT

N = 6

Expands the range of the TransferFunction. The first value within a pair is associated with TransferBlack and the second is associated with TransferWhite. The ordering of pairs is the same as for pixel components of the PhotometricInterpretation type. By default, the TransferFunction is defined over a range from a minimum intensity, 0 or nominal black, to a maximum intensity, $(1 \ll \text{BitsPerSample}) - 1$ or nominal white. Kodak PhotoYCC uses an extended range TransferFunction in order to describe highlights, saturated colors and shadow detail beyond this range. The TransferRange expands the TransferFunction to support these values. It is defined only for RGB and YCbCr PhotometricInterpretations.

After ReferenceBlackWhite and/or YCbCr decoding has taken place, an RGB value can be represented as a real number. It is then rounded to create an index into the TransferFunction table. In the absence of a TransferRange tag, or if the tag has the default values, the rounded value is an index and the normalized intensity value is:

```
index = (int) (value + (value < 0.0? -0.5 : 0.5));
intensity = TF[index] / 65535;
```

If the TransferRange tag is present and has non-default values, it provides an offset to be used with the rounded index. It also describes a scaling. The normalized intensity value is:

```
index = (int) (value + (value < 0.0? -0.5 : 0.5));
intensity = (TF[index + TransferRange[Black]] -
             TF[TransferRange[Black]])
            / (TF[TransferRange[White]] - TF[TransferRange[Black]]);
```

An application can write a TransferFunction with a non-default TransferRange as follows:

```
black_offset = scale_factor * Transfer(-TransferRange[Black]) /
              (TransferRange[White] - TransferRange[Black]);
for (i = 0; i < (1 << BitsPerSample); i++)
    TF[i] = floor(0.5 - black_offset + scale_factor
                 * Transfer((i - TransferRange[Black])
                           / (TransferRange[White] - TransferRange[Black])));
```

The TIFF writer chooses `scale_factor` such that the TransferFunction fits into a 16-bit unsigned short, and chooses the TransferRange so that the most important part of the TransferFunction fits into the table.

Default is [0, NV, 0, NV, 0, NV] where $NV = (1 \ll \text{BitsPerSample}) - 1$.

ReferenceBlackWhite

Tag = 532 (214.H)

Type = RATIONAL

N = 6

Specifies a pair of headroom and footroom image data values (codes) for each pixel component. The first component code within a pair is associated with ReferenceBlack, and the second is associated with ReferenceWhite. The ordering of pairs is the same as those for pixel components of the PhotometricInterpretation type. ReferenceBlackWhite can be applied to images with a PhotometricInterpretation value of RGB or YCbCr. ReferenceBlackWhite is not used with other PhotometricInterpretation values.

Computer graphics commonly places black and white at the extremities of the binary representation of image data; for example, black at code 0 and white at code 255. In other disciplines, such as printing, film, and video, there are practical reasons to provide footroom codes below ReferenceBlack and headroom codes above ReferenceWhite.

In film applications, they correspond to the densities Dmax and Dmin. In video applications, ReferenceBlack corresponds to 7.5 IRE and 0 IRE in systems with and without setup respectively, and ReferenceWhite corresponds to 100 IRE units.

Using YCbCr (See Section 21) and the CCIR Recommendation 601.1 video standard as an example, code 16 represents ReferenceBlack, and code 235 represents ReferenceWhite for the luminance component (Y). For the chrominance components, Cb and Cr, code 128 represents ReferenceBlack, and code 240 represents ReferenceWhite. With Cb and Cr, the ReferenceWhite value is used to code reference blue and reference red respectively.

The full range component value is converted from the code by:

$$\text{FullRangeValue} = (\text{code} - \text{ReferenceBlack}) * \text{CodingRange} / (\text{ReferenceWhite} - \text{ReferenceBlack});$$

The code is converted from the full-range component value by:

$$\text{code} = (\text{FullRangeValue} * (\text{ReferenceWhite} - \text{ReferenceBlack}) / \text{CodingRange}) + \text{ReferenceBlack};$$

For RGB images and the Y component of YCbCr images, CodingRange is defined as:

$$\text{CodingRange} = 2 ** \text{BitsPerSample} - 1;$$

For the Cb and Cr components of YCbCr images, CodingRange is defined as:

$$\text{CodingRange} = 127;$$

For RGB images, in the default special case of no headroom or footroom, this conversion can be skipped because the scaling multiplier equals 1.0 and the value equals the code.

For YCbCr images, in the case of no headroom or footroom, the conversion for Y can be skipped because the value equals the code. For Cb and Cr, ReferenceBlack must still be subtracted from the code. In the general case, the scaling multiplication for the Cb and Cr component codes can be factored into the YCbCr transform matrix.

Useful ReferenceBlackWhite values for YCbCr images are:

$$[0/1, 255/1, 128/1, 255/1, 128/1, 255/1]$$

no headroom/footroom

$$[15/1, 235/1, 128/1, 240/1, 128/1, 240/1]$$

CCIR Recommendation 601.1 headroom/footroom

Useful ReferenceBlackWhite values for BitsPerSample = 8,8,8 Class R images are:

[0/1, 255/1,0/1, 255/1, 0/1, 255/1]

no headroom/footroom

[16/1, 235/1, 16/1, 235/1, 16/1, 235/1]

CCIR Recommendation 601.1 headroom/footroom

Default is [0,NV/1, 0/1, NV/1, 0/1, NV/1] where $NV = 2^{**} \text{BitsPerSample} - 1$.

References

- [1] *The Reproduction of Colour in Photography, Printing and Television*, R. W. G. Hunt, Fountain Press, Tolworth, England,1987.
- [2] *Principles of Color Technology*, Billmeyer and Saltzman, Wiley-Interscience, New York, 1981.
- [3] *Colorimetric Properties of Video Displays*, William Cowan, University of Waterloo, Waterloo, Canada, 1989.
- [4] *TIFF Color Appearance Guidelines*, Dave Farber, Eastman Kodak Company, Rochester, New York.

Section 21: YC_bC_r Images

Introduction

Digitizers of video sources that create RGB data are becoming more capable and less expensive. The RGB color space is adequate for this purpose. However, for both digital video and image compression applications a color difference color space is needed. The television industry depends on YC_bC_r for digital video. For image compression, subsampling the chrominance components allows for greater compression. TIFF YC_bC_r (which we shall call *Class Y*) supports these images and applications.

Class Y is based on CCIR Recommendation 601-1, “Encoding Parameters of Digital Television for Studios.” Class Y also has parameters that allow the description of related standards such as CCIR Recommendation 709 and technological variations such as component-sample positioning.

YC_bC_r is a distinct Photometric Interpretation type. RGB pixels are converted to and from YC_bC_r for storage and display.

Class Y defines the following fields:

YC _b C _r Coefficients	transformation from RGB to YC _b C _r
YC _b C _r SubSampling	subsampling of the chrominance components
YC _b C _r Positioning	positioning of chrominance component samples relative to the luminance samples

In addition, ReferenceBlackWhite, which specifies coding range expansion, is required by Class Y. See Section 20.

Class Y YC_bC_r images have three components: Y, the luminance component, and C_b and C_r, two chrominance components. Class Y uses the international standard notation YC_bC_r for color-difference component coding. This is often incorrectly called YUV, which properly applies only to composite coding.

The transformations between YC_bC_r and RGB are linear transformations of uninterpreted RGB sample data, typically gamma-corrected values. The YC_bC_rCoefficients field describes the parameters of this transformation.

Another feature of Class Y comes from subsampling the chrominance components. A Class Y image can be compressed by reducing the spatial resolution of chrominance components. This takes advantage of the relative insensitivity of the human visual system to chrominance detail. The YC_bC_rSubSampling field describes the degree of subsampling which has taken place.

When a Class Y image is subsampled, each C_b and C_r sample is associated with a group of luminance samples. The YC_bC_rPositioning field describes the position of the chrominance component samples relative to the group of luminance samples: centered or cosited.

Class Y requires use of the ReferenceBlackWhite field. This field expands the coding range by describing the reference black and white values for the different components that allow headroom and footroom for digital video images. Since the

default for ReferenceBlackWhite is inappropriate for Class Y, it must be used explicitly.

At first, it might seem that the information conveyed by Class Y and the RGB Colorimetry section is redundant. However, decoding YC_bC_r to RGB primaries requires the YC_bC_r fields, and interpretation of the resulting RGB primaries requires the colorimetry and transfer function information. See the RGB Colorimetry section for details.

Extensions to Existing Fields

Class Y images use a distinct PhotometricInterpretation Field value:

PhotometricInterpretation

Tag = 262 (106.H)

Type = SHORT

N = 1

This Field indicates the color space of the image. The new value is:

6 = YC_bC_r

A value of 6 indicates that the image data is in the YC_bC_r color space. TIFF uses the international standard notation YC_bC_r for color-difference sample coding. Y is the luminance component. C_b and C_r are the two chrominance components. RGB pixels are converted to and from YC_bC_r form for storage and display.

Fields Defined in Class Y

YC_bC_r Coefficients

Tag = 529 (211.H)

Type = RATIONAL

N = 3

The transformation from RGB to YC_bC_r image data. The transformation is specified as three rational values that represent the coefficients used to compute luminance, Y.

The three rational coefficient values, *LumaRed*, *LumaGreen* and *LumaBlue*, are the proportions of red, green, and blue respectively in luminance, Y.

Y, C_b , and C_r may be computed from RGB using the luminance coefficients specified by this field as follows:

$$Y = (LumaRed * R + LumaGreen * G + LumaBlue * B)$$

$$C_b = (B - Y) / (2 - 2 * LumaBlue)$$

$$C_r = (R - Y) / (2 - 2 * LumaRed)$$

R, G, and B may be computed from YC_bC_r as follows:

$$R = C_r * (2 - 2 * LumaRed) + Y$$

$$G = (Y - LumaBlue * B - LumaRed * R) / LumaGreen$$

$$B = C_b * (2 - 2 * LumaBlue) + Y$$

In disciplines such as printing, film, and video, there are practical reasons to provide footroom codes below the ReferenceBlack code and headroom codes above ReferenceWhite code. In such cases the values of the transformation matrix used to convert from YC_bC_r to RGB must be multiplied by a scale factor to produce full-range RGB values. These scale factors depend on the reference ranges specified by the ReferenceBlackWhite field. See the ReferenceBlackWhite and TransferFunction fields for more details.

The values coded by this field will typically reflect the transformation specified by a standard for YC_bC_r encoding. The following table contains examples of commonly used values.

Standard	<i>LumaRed</i>	<i>LumaGreen</i>	<i>LumaBlue</i>
CCIR Recommendation 601-1	299 / 1000	587 / 1000	114 / 1000
CCIR Recommendation 709	2125 / 10000	7154 / 10000	721 / 10000

The default values for this field are those defined by CCIR Recommendation 601-1: 299/1000, 587/1000 and 114/1000, for *LumaRed*, *LumaGreen* and *LumaBlue*, respectively.

YC_bC_rSubSampling

Tag = 530 (212.H)

Type = SHORT

N = 2

Specifies the subsampling factors used for the chrominance components of a YC_bC_r image. The two fields of this field, *YC_bC_rSubsampleHoriz* and *YC_bC_rSubsampleVert*, specify the horizontal and vertical subsampling factors respectively.

The two fields of this field are defined as follows:

Short 0: *YC_bC_rSubsampleHoriz*:

- 1 = ImageWidth of this chroma image is equal to the ImageWidth of the associated luma image.
- 2 = ImageWidth of this chroma image is half the ImageWidth of the associated luma image.
- 4 = ImageWidth of this chroma image is one-quarter the ImageWidth of the associated luma image.

Short 1: *YC_bC_rSubsampleVert*:

- 1 = ImageLength (height) of this chroma image is equal to the ImageLength of the associated luma image.

- 2 = ImageLength (height) of this chroma image is half the ImageLength of the associated luma image.
- 4 = ImageLength (height) of this chroma image is one-quarter the ImageLength of the associated luma image.

Both C_b and C_r have the same subsampling ratio. Also, $YC_bC_rSubsampleVert$ shall always be less than or equal to $YC_bC_rSubsampleHoriz$.

ImageWidth and ImageLength are constrained to be integer multiples of $YC_bC_rSubsampleHoriz$ and $YC_bC_rSubsampleVert$ respectively. TileWidth and TileLength have the same constraints. RowsPerStrip must be an integer multiple of $YC_bC_rSubsampleVert$.

The default values of this field are [2, 2].

YC_bC_rPositioning

Tag = 531 (213.H)

Type = SHORT

N = 1

Specifies the positioning of subsampled chrominance components relative to luminance samples.

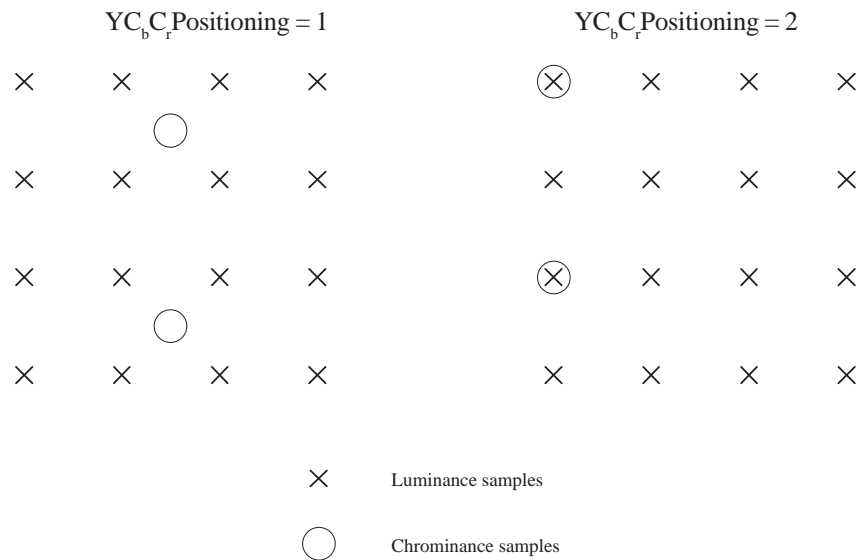
Specification of the spatial positioning of pixel samples relative to the other samples is necessary for proper image post processing and accurate image presentation. In Class Y files, the position of the subsampled chrominance components are defined with respect to the luminance component. Because components 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 luminance component. The horizontal and vertical offsets of the first chrominance sample are denoted Xoffset[0,0] and Yoffset[0,0] respectively. Xoffset[0,0] and Yoffset[0,0] are defined in terms of the number of samples in the luminance component.

The values for this field are defined as follows:

Tag value	YC_bC_r Positioning	X and Y offsets of first chrominance sample
1	centered	Xoffset[0,0] = $ChromaSubsampleHoriz / 2 - 0.5$ Yoffset[0,0] = $ChromaSubsampleVert / 2 - 0.5$
2	cosited	Xoffset[0,0] = 0 Yoffset[0,0] = 0

Field value 1 (centered) must be specified for compatibility with industry standards such as PostScript Level 2 and QuickTime. Field value 2 (cosited) must be specified for compatibility with most digital video standards, such as CCIR Recommendation 601-1.

As an example, for $ChromaSubsampleHoriz = 4$ and $ChromaSubsampleVert = 2$, the centers of the samples are positioned as illustrated below:



Luminance samples

Chrominance samples

Proper subsampling of the chrominance components incorporates an anti-aliasing filter that reduces the spectral bandwidth of the full-resolution samples. The type of filter used for subsampling determines the value of the YCbCrPositioning field.

For YCbCrPositioning = 1 (centered), subsampling of the chrominance components 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).

For YCbCrPositioning = 2 (cosited), subsampling of the chrominance components can easily be accomplished using a symmetrical digital filter with an odd number of taps. A commonly used filter for 2:1 subsampling utilizes three taps (1/4,1/2,1/4).

The default value of this field is 1.

Ordering of Component Samples

This section defines the ordering convention used for Y, C_b, and C_r component samples when the PlanarConfiguration field value = 1 (interleaving). For PlanarConfiguration = 2, component samples are stored as 3 separate planes, and the ordering is the same as that used for other PhotometricInterpretation field values.

For PlanarConfiguration = 1, the component sample order is based on the subsampling factors, *ChromaSubsampleHoriz* and *ChromaSubsampleVert*, defined by the YCbCrSubSampling field. The image data within a TIFF file is comprised of one or more “data units”, where a data unit is defined to be a sequence of samples:

- one or more Y samples
- a C_b sample
- a C_r sample

The Y samples within a data unit are specified as a two-dimensional array having *ChromaSubsampleVert* rows of *ChromaSubsampleHoriz* samples.

Expanding on the example in the previous section, consider a $Y C_b C_r$ image having $ChromaSubsampleHoriz = 4$ and $ChromaSubsampleVert = 2$:

Y component								Cb component		Cr component	
Y00	Y01	Y02	Y03	Y04	Y05			Cb00		Cr00	
Y10	Y11	Y12	Y13								

For $PlanarConfiguration = 1$, the sample order is:

$$Y_{00}, Y_{01}, Y_{02}, Y_{03}, Y_{10}, Y_{11}, Y_{12}, Y_{13}, Cb_{00}, Cr_{00}, Y_{04}, Y_{05} \dots$$

Minimum Requirements for YCbCr Images

In addition to satisfying the general Baseline TIFF requirements, a YCbCr file must have the following characteristics:

- $SamplesPerPixel = 3$. SHORT. Three components representing Y, Cb and Cr.
- $BitsPerSample = 8,8,8$. SHORT.
- $Compression = none (1), LZW (5) \text{ or } JPEG (6)$. SHORT.
- $PhotometricInterpretation = Y C_b C_r (6)$. SHORT.
- $ReferenceBlackWhite = 6$ RATIONALS. Specify the reference values for black and white.

If the conversion from RGB is not according to CCIR Recommendation 601-1, code $Y C_b C_r$ Coefficients.

Section 22: JPEG Compression

Introduction

Image compression reduces the storage requirements of pictorial data. In addition, it reduces the time required for access to, communication with, and display of images. To address the standardization of compression techniques an international standards group was formed: the Joint Photographic Experts Group (JPEG). JPEG has as its objective to create a joint ISO/CCITT standard for continuous tone image compression (color and grayscale).

JPEG decided that because of the broad scope of the standard, no one algorithmic procedure was able to satisfy the requirements of all applications. It was decided to specify different algorithmic processes, where each process is targeted to satisfy the requirements of a class of applications. Thus, the JPEG standard became a “toolkit” whereby the particular algorithmic “tools” are selected according to the needs of the application environment.

The algorithmic processes fall into two classes: lossy and lossless. Those based on the Discrete Cosine Transform (DCT) are lossy and typically provide for substantial compression without significant degradation of the reconstructed image with respect to the source image.

The simplest DCT-based coding process is the baseline process. It provides a capability that is sufficient for most applications. There are additional DCT-based processes that extend the baseline process to a broader range of applications.

The second class of coding processes is targeted for those applications requiring lossless compression. The lossless processes are not DCT-based and are utilized independently of any of the DCT-based processes.

This Section describes the JPEG baseline, the JPEG lossless processes, and the extensions to TIFF defined to support JPEG compression.

JPEG Baseline Process

The baseline process is a DCT-based algorithm that compresses images having 8 bits per component. The baseline process operates only in sequential mode. In sequential mode, the image is processed from left to right and top to bottom in a single pass by compressing the first row of data, followed by the second row, and continuing until the end of image is reached. Sequential operation has minimal buffering requirements and thus permits inexpensive implementations.

The JPEG baseline process is an algorithm which inherently introduces error into the reconstructed image and cannot be utilized for lossless compression. The algorithm accepts as input only those images having 8 bits per component. Images with fewer than 8 bits per component may be compressed using the baseline process algorithm by left justifying each input component within a byte before compression.

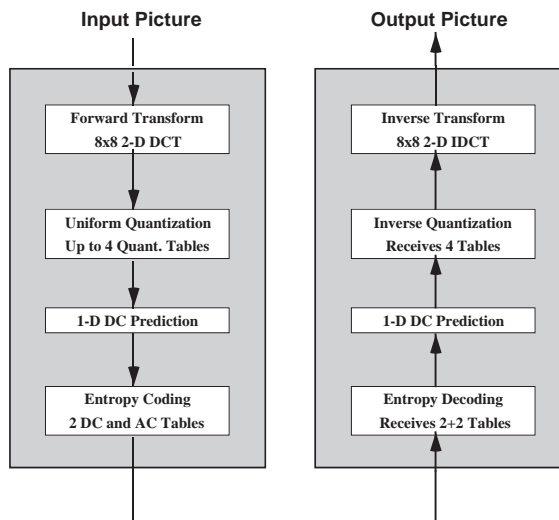


Figure 1. Baseline Process Encoder and Decoder

A functional block diagram of the Baseline encoding and decoding processes is contained in Figure 1. Encoder operation consists of dividing each component of the input image into 8x8 blocks, performing the two-dimensional DCT on each block, quantizing each DCT coefficient uniformly, subtracting the quantized DC coefficient from the corresponding term in the previous block, and then entropy coding the quantized coefficients using variable length codes (VLCs). Decoding is performed by inverting each of the encoder operations in the reverse order.

The DCT

Before performing the forward DCT, input pixels are level-shifted so that they range from -128 to +127. Blocks of 8x8 pixels are transformed with the two-dimensional 8x8 DCT:

$$F(u,v) = \frac{1}{4} C(u)C(v) \int f(x,y) \cos \frac{(2x+1)u}{16} \cos \frac{(2y+1)v}{16}$$

and blocks are inverse transformed by the decoder with the Inverse DCT:

$$f(x,y) = \frac{1}{4} C(u)C(v) \int F(u,v) \cos \frac{(2x+1)u}{16} \cos \frac{(2y+1)v}{16}$$

with $u, v, x, y = 0, 1, 2, \dots, 7$

where $x, y =$ spatial coordinates in the pel domain

$u, v =$ coordinates in the transform domain

$$C(u), C(v) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u, v = 0 \\ 1 & \text{otherwise} \end{cases}$$

Although the exact method for computation of the DCT and IDCT is not subject to standardization and will not be specified by JPEG, it is probable that JPEG will adopt DCT-conformance specifications that designate the accuracy to which the DCT must be computed. The DCT-conformance specifications will assure that any two JPEG implementations will produce visually-similar reconstructed images.

Quantization

The coefficients of the DCT are quantized to reduce their magnitude and increase the number of zero-value coefficients. The DCT coefficients are independently quantized by uniform quantizers. A uniform quantizer divides the real number line into steps of equal size, as shown in Figure 2. The quantization step-size applied to each coefficient is determined from the contents of a 64-element quantization table.

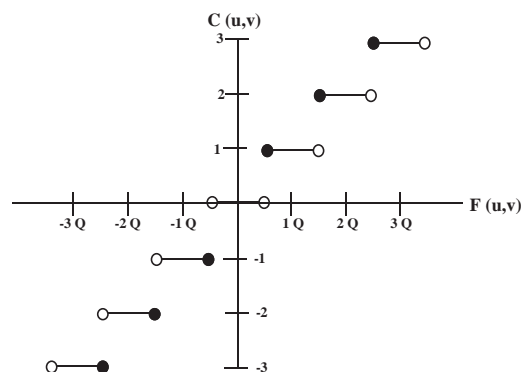


Figure 2. Uniform Quantization

The baseline process provides for up to 4 different quantization tables to be defined and assigned to separate interleaved components within a single scan of the input image. Although the values of each quantization table should ideally be determined through rigorous subjective testing which estimates the human psycho-visual thresholds for each DCT coefficient and for each color component of the input image, JPEG has developed quantization tables which work well for CCIR 601 resolution images and has published these in the informational section of the proposed standard.

DC Prediction

The DCT coefficient located in the upper-left corner of the transformed block represents the average spatial intensity of the block and is referred to as the “DC coefficient”. After the DCT coefficients are quantized, but before they are entropy coded, DC prediction is performed. DC prediction simply means that the DC term of the previous block is subtracted from the DC term of the current block prior to encoding.

Zig-Zag Scan

Prior to entropy coding, the DCT coefficients are ordered into a one-dimensional sequence according to a “zig-zag” scan. The DC coefficient is coded first, followed by AC coefficient coding, proceeding in the order illustrated in Figure 3.

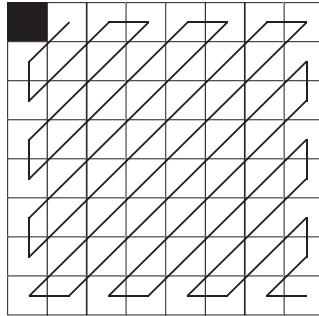


Figure 3. Zig-Zag Scan of DCT Coefficients

Entropy Coding

The quantized DCT coefficients are further compressed using entropy coding. The baseline process performs entropy coding using variable length codes (VLCs) and variable length integers (VLIs).

VLCs, commonly known as Huffman codes, compress data symbols by creating shorter codes to represent frequently-occurring symbols and longer codes for occasionally-occurring symbols. One reason for using VLCs is that they are easily implemented by means of lookup tables.

Separate code tables are provided for the coding of DC and AC coefficients. The following paragraphs describe the respective coding methods used for coding DC and AC coefficients.

DC Coefficient Coding

DC prediction produces a “differential DC coefficient” that is typically small in magnitude due to the high correlation of neighboring DC coefficients. Each differential DC coefficient is encoded by a VLC which represents the number of significant bits in the DC term followed by a VLI representing the value itself. The VLC is coded by first determining the number of significant bits, SSSS, in the differential DC coefficient through the following table:

SSSS	Differential DC Value
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	-1023..-512, 512..1023
11	-2047..-1024, 1024..2047
12	-4095..-2048, 2048..4095

SSSS is then coded from the selected DC VLC table. The VLC is followed by a VLI having SSSS bits that represents the value of the differential DC coefficient itself. If the coefficient is positive, the VLI is simply the low-order bits of the coefficient. If the coefficient is negative, then the VLI is the low-order bits of the coefficient-1.

AC Coefficient Coding

In a similar fashion, AC coefficients are coded with alternating VLC and VLI codes. The VLC table, however, is a two-dimensional table that is indexed by a composite 8-bit value. The lower 4 bits of the 8-bit value, i.e. the column index, is the number of significant bits, SSSS, of a non-zero AC coefficient. SSSS is computed through the same table as that used for coding the DC coefficient. The higher-order 4 bits, the row index, is the number of zero coefficients, NNNN, that precede the non-zero AC coefficient. The first column of the two-dimensional coding table contains codes that represent control functions. Figure 4 illustrates the general structure of the AC coding table.

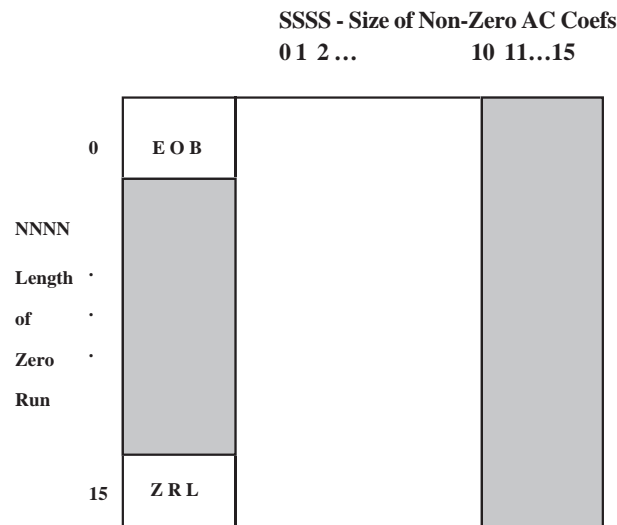


Figure 4. 2-D Run-Size Value Array for AC Coefs
 The shaded portions are undefined in the baseline process

The flow chart in Figure 5 specifies the AC coefficient coding procedure. AC coefficients are coded by traversing the block in the zig-zag sequence and count-

ing the number of zero coefficients until a non-zero AC coefficient is encountered. If the count of consecutive zero coefficients exceeds 15, then a ZRL code is coded and the zero run-length count is reset. When a non-zero AC coefficient is found, the number of significant bits in the non-zero coefficient, SSSS, is combined with the zero run-length that precedes the coefficient, NNNN, to form an index into the two-dimensional VLC table. The selected VLC is then coded. The VLC is followed by a VLI that represents the value of the AC coefficient. This process is repeated until the end of the block is reached. If the last AC coefficient is zero, then an End of Block (EOB) VLC is encoded.

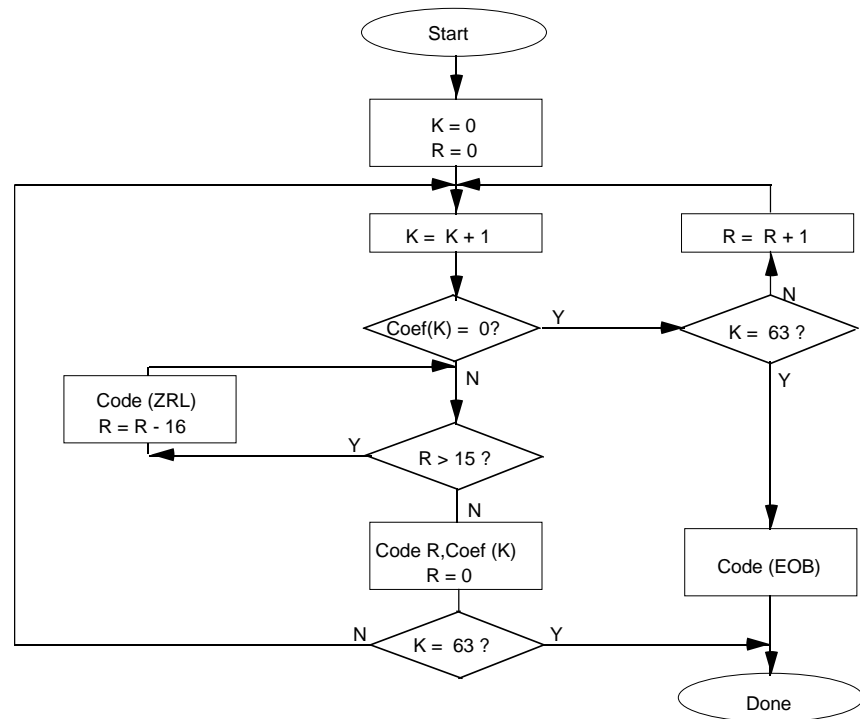


Figure 5. Encoding Procedure for AC Coeffs

JPEG Lossless Processes

The JPEG lossless coding processes utilize a spatial-prediction algorithm based upon a two-dimensional Differential Pulse Code Modulation (DPCM) technique. They are compatible with a wider range of input pixel precision than the DCT-based algorithms (2 to 16 bits per component). Although the primary motivation for specifying a spatial algorithm is to provide a method for lossless compression, JPEG allows for quantization of the input data, resulting in lossy compression and higher compression rates.

Although JPEG provides for use of either the Huffman or Arithmetic entropy-coding models by the processes for lossless coding, only the Huffman coding model is supported by this version of TIFF. The following is a brief overview of the lossless process with Huffman coding.

Control Structure

Much of the control structure developed for the sequential DCT procedures is also used for sequential lossless coding. Either interleaved or non-interleaved data ordering may be used.

Coding Model

The coding model developed for coding the DC coefficients of the DCT is extended to allow a number of one-dimensional and two-dimensional predictors for the lossless coding function. Each component uses an independent predictor.

Prediction

Figure 6 shows the relationship between the neighboring values used for prediction and the sample being coded.

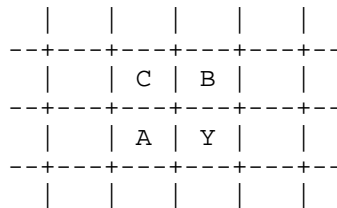


Figure 6. Relationship between sample and prediction samples

Y is the sample to be coded and A, B, and C are the samples immediately to the left, immediately above, and diagonally to the left and above.

The allowed predictors are listed in the following table.

Selection-value	Prediction
0	no prediction (differential coding)
1	A
2	B
3	C
4	$A+B-C$
5	$A+((B-C)/2)$
6	$B+((A-C)/2)$
7	$(A+B)/2$

Selection-value 0 shall only be used for differential coding in the hierarchical mode. Selections 1, 2 and 3 are one-dimensional predictors and selections 4, 5, 6, and 7 are two dimensional predictors. The divide by 2 in the prediction equations is done by a arithmetic-right-shift of the integer values.

The difference between the prediction value and the input is calculated modulo $2^{**}16$. Therefore, the prediction can also be treated as a modulo $2^{**}16$ value. In the decoder the difference is decoded and added, modulo $2^{**}16$, to the prediction.

Huffman Coding of the Prediction Error

The Huffman coding procedures defined for coding the DC coefficients are used to code the modulo $2^{**}16$ differences. The table for DC coding is extended to 17 entries that allows for coding of the modulo $2^{**}16$ differences.

Point Transformation Prior to Lossless Coding

For the lossless processes only, the input image data may optionally be scaled (quantized) prior to coding by specifying a nonzero value in the point transformation parameter. Point transformation is defined to be division by a power of 2.

If the point transformation field is nonzero for a component, a point transformation of the input is performed prior to the lossless coding. The input is divided by $2^{**}Pt$, where Pt is the value of the point transform signaling field. The output of the decoder is rescaled to the input range by multiplying by $2^{**}Pt$. Note that the scaling of input and output can be performed by arithmetic shifts.

Overview of the JPEG Extension to TIFF

In extending the TIFF definition to include JPEG compressed data, it is necessary to note the following:

- JPEG is effective only on continuous-tone color spaces:

Grayscale (Photometric Interpretation = 1)

RGB (Photometric Interpretation = 2)

CMYK (Photometric Interpretation = 5) (See the CMYK Images section.)

$YCbCr$ (Photometric Interpretation = 6) (See the YCbCr images section.)

- Color conversion to $YCbCr$ is often used as part of the compression process because the chrominance components can be subsampled and compressed to a greater degree without significant visual loss of quality. Fields are defined to describe how this conversion has taken place and the degree of subsampling employed (see the YCbCr Images section).
- New fields have been defined to specify the JPEG parameters used for compression and to allow quantization tables and Huffman code tables to be incorporated into the TIFF file.

- TIFF is compatible with compressed image data that conforms to the syntax of the JPEG interchange format for compressed image data. Fields are defined that may be utilized to facilitate conversion from TIFF to interchange format.
- The PlanarConfiguration Field is used to specify whether or not the compressed data is interleaved as defined by JPEG. For any of the JPEG DCT-based processes, the interleaved data units are coded 8x8 blocks rather than component samples.
- Although JPEG codes consecutive image blocks in a single contiguous bitstream, it is extremely useful to employ the concept of tiles in an image. The TIFF Tiles section defines some new fields for tiles. These fields should be stored in place of the older fields for strips. The concept of tiling an image in both dimensions is important because JPEG hardware may be limited in the size of each block that is handled.
- Note that the nomenclature used in the TIFF specification is different from the JPEG Draft International Standard (ISO DIS 10918-1) in some respects. The following terms should be equated when reading this Section:

TIFF name	JPEG DIS name
ImageWidth	Number of Pixels
ImageLength	Number of Lines
SamplesPerPixel	Number of Components
JPEGQTable	Quantization Table
JPEGDCTable	Huffman Table for DC coefficients
JPEGACTable	Huffman Table for AC coefficients

Strips and Tiles

The JPEG extension to TIFF has been designed to be consistent with the existing TIFF strip and tile structures and to allow quick conversion to and from the stream-oriented compressed image format defined by JPEG.

Compressed images conforming to the syntax of the JPEG interchange format can be converted to TIFF simply by defining a single strip or tile for the entire image and then concatenating the TIFF image description fields to the JPEG compressed image data. The strip or tile offset field points directly to the start of the entropy coded data (not to a JPEG marker).

Multiple strips or tiles are supported in JPEG compressed images using restart markers. Restart markers, inserted periodically into the compressed image data, delineate image segments known as restart intervals. At the start of each restart interval, the coding state is reset to default values, allowing every restart interval to be decoded independently of previously decoded data. TIFF strip and tile offsets shall always point to the start of a restart interval. Equivalently, each strip or

tile contains an integral number of restart intervals. Restart markers need not be present in a TIFF file; they are implicitly coded at the start of every strip or tile.

To maximize interchangeability of TIFF files with other formats, a restriction is placed on tile height for files containing JPEG-compressed image data conforming to the JPEG interchange format syntax. The restriction, imposed only when the tile width is shorter than the image width and when the JPEGInterchangeFormat Field is present and non-zero, states that the tile height must be equal to the height of one JPEG Minimum Coded Unit (MCU). This restriction ensures that TIFF files may be converted to JPEG interchange format without undergoing decompression.

Extensions to Existing Fields

Compression

Tag = 259 (103.H)

Type = SHORT

N = 1

This Field indicates the type of compression used. The new value is:

6 = JPEG

JPEG Fields

JPEGProc

Tag = 512 (200.H)

Type = SHORT

N = 1

This Field indicates the JPEG process used to produce the compressed data. The values for this field are defined to be consistent with the numbering convention used in ISO DIS 10918-2. Two values are defined at this time.

- 1= Baseline sequential process
- 14= Lossless process with Huffman coding

When the lossless process with Huffman coding is selected by this Field, the Huffman tables used to encode the image are specified by the JPEGDCTables field, and the JPEGACTables field is not used.

Values indicating JPEG processes other than those specified above will be defined in the future.

Not all of the fields described in this section are relevant to the JPEG process selected by this Field. The following table specifies the fields that are applicable to each value defined by this Field.

Tag Name	JPEGProc =1	JPEGProc =14
JPEGInterchangeFormat	X	X
JPEGInterchangeFormatLength	X	X
JPEGRestartInterval	X	X
JPEGLosslessPredictors		X
JPEGPointTransforms		X
JPEGQTables	X	
JPEGDCTables	X	X
JPEGACTables	X	

This Field is mandatory whenever the Compression Field is JPEG (no default).

JPEGInterchangeFormat

Tag = 513 (201.H)

Type = LONG

N = 1

This Field indicates whether a JPEG interchange format bitstream is present in the TIFF file. If a JPEG interchange format bitstream is present, then this Field points to the Start of Image (SOI) marker code.

If this Field is zero or not present, a JPEG interchange format bitstream is not present.

JPEGInterchangeFormatLength

Tag = 514 (202.H)

Type = LONG

N = 1

This Field indicates the length in bytes of the JPEG interchange format bitstream. This Field is useful for extracting the JPEG interchange format bitstream without parsing the bitstream.

This Field is relevant only if the JPEGInterchangeFormat Field is present and is non-zero.

JPEGRestartInterval

Tag = 515 (203.H)

Type = SHORT

N = 1

This Field indicates the length of the restart interval used in the compressed image data. The restart interval is defined as the number of Minimum Coded Units (MCUs) between restart markers.

Restart intervals are used in JPEG compressed images to provide support for multiple strips or tiles. At the start of each restart interval, the coding state is reset to default values, allowing every restart interval to be decoded independently of previously decoded data. TIFF strip and tile offsets shall always point to the start of a restart interval. Equivalently, each strip or tile contains an integral number of restart intervals. Restart markers need not be present in a TIFF file; they are implicitly coded at the start of every strip or tile.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more information about the restart interval and restart markers.

If this Field is zero or is not present, the compressed data does not contain restart markers.

JPEGLosslessPredictors

Tag = 517 (205.H)

Type = SHORT

N = SamplesPerPixel

This Field points to a list of lossless predictor-selection values, one per component.

The allowed predictors are listed in the following table.

Selection-value	Prediction
1	A
2	B
3	C
4	A+B-C
5	$A + ((B - C) / 2)$
6	$B + ((A - C) / 2)$
7	$(A + B) / 2$

A, B, and C are the samples immediately to the left, immediately above, and diagonally to the left and above the sample to be coded, respectively.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details.

This Field is mandatory whenever the JPEGProc Field specifies one of the lossless processes (no default).

JPEGPointTransforms

Tag = 518 (206.H)

Type = SHORT

N = SamplesPerPixel

This Field points to a list of point transform values, one per component. This Field is relevant only for lossless processes.

If the point transformation value is nonzero for a component, a point transformation of the input is performed prior to the lossless coding. The input is divided by 2^{Pt} , where Pt is the point transform value. The output of the decoder is rescaled to the input range by multiplying by 2^{Pt} . Note that the scaling of input and output can be performed by arithmetic shifts.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details. The default value of this Field is 0 for each component (no scaling).

JPEGQTables

Tag = 519 (207.H)

Type = LONG

N = SamplesPerPixel

This Field points to a list of offsets to the quantization tables, one per component. Each table consists of 64 BYTES (one for each DCT coefficient in the 8x8 block). The quantization tables are stored in zigzag order.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details.

It is strongly recommended that, within the TIFF file, each component be assigned separate tables. This Field is mandatory whenever the JPEGProc Field specifies a DCT-based process (no default).

JPEGDCTables

Tag = 520 (208.H)

Type = LONG

N = SamplesPerPixel

This Field points to a list of offsets to the DC Huffman tables or the lossless Huffman tables, one per component.

The format of each table is as follows:

16 BYTES of “BITS”, indicating the number of codes of lengths 1 to 16;

Up to 17 BYTES of “VALUES”, indicating the values associated with those codes, in order of length.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details.

It is strongly recommended that, within the TIFF file, each component be assigned separate tables. This Field is mandatory for all JPEG processes (no default).

JPEGACTables

Tag = 521 (209.H)

Type = LONG

N = SamplesPerPixel

This Field points to a list of offsets to the Huffman AC tables, one per component. The format of each table is as follows:

16 BYTES of “BITS”, indicating the number of codes of lengths 1 to 16;

Up to 256 BYTES of “VALUES”, indicating the values associated with those codes, in order of length.

See the JPEG Draft International Standard (ISO DIS 10918-1) for more details.

It is strongly recommended that, within the TIFF file, each component be assigned separate tables. This Field is mandatory whenever the JPEGProc Field specifies a DCT-based process (no default).

Minimum Requirements for TIFF with JPEG Compression

The table on the following page shows the minimum requirements of a TIFF file that uses tiling and contains JPEG data compressed with the Baseline process.

Tag = NewSubFileType (254) Type = Long Length = 1 Value = 0	Single image
Tag = ImageWidth (256) Type = Long Length = 1 Value = ?	
Tag = ImageLength (257) Type = Long Length = 1 Value = ?	
Tag = BitsPerSample (258) Type = Short Length = SamplesPerPixel Value = ?	8 : Monochrome 8,8,8 : RGB 8,8,8 : YCbCr 8,8,8,8 : CMYK
Tag = Compression (259) Type = Long Length = 1 Value = 6	6 : JPEG compression
Tag = PhotometricInterpretation (262) Type = Short Length = 1 Value = ?	0,1 : Monochrome 2 : RGB 5 : CMYK 6 : YCbCr
Tag = SamplesPerPixel (277) Type = Short Length = 1 Value = ?	1 : Monochrome 3 : RGB 3 : YCbCr 4 : CMYK
Tag = XResolution (282) Type = Rational Length = 1 Value = ?	
Tag = YResolution (283) Type = Rational Length = 1 Value = ?	
Tag = PlanarConfiguration (284) Type = Short Length = 1 Value = ?	1 : Block interleaved 2 : Not interleaved
Tag = ResolutionUnit (296) Type = Short Length = 1 Value = ?	
Tag = TileWidth (322) Type = Short Length = 1 Value = ?	Multiple of 8
Tag = TileLength (323) Type = Short Length = 1 Value = ?	Multiple of 8
Tag = TileOffsets (324) Type = Long Length = Number of tiles Value = ?	
Tag = TileByteCounts (325) Type = Long Length = Number of tiles Value = ?	
Tag = JPEGProc (512) Type = Short Length = 1 Value = ?	1 : Baseline process
Tag = JPEGQTables (519) Type = Long Length = SamplesPerPixel Value = ?	Offsets to tables
Tag = JPEGDCTables (520) Type = Long Length = SamplesPerPixel Value = ?	Offsets to tables
Tag = JPEGACTables (521) Type = Long Length = SamplesPerPixel Value = ?	Offsets to tables

References

[1] Wallace, G., "Overview of the JPEG Still Picture Compression Algorithm", Electronic Imaging East '90.

[2] ISO/IEC DIS 10918-1, "Digital Compression and Coding of Continuous-tone Still Images", Sept. 1991.

Section 23: CIE L*a*b* Images

What is CIE L*a*b*?

CIE L*a*b* is a color space that is colorimetric, has separate lightness and chroma channels, and is approximately perceptually uniform. It has excellent applicability for device-independent manipulation of continuous tone images. These attributes make it an excellent choice for many image editing functions.

1976 CIE L*a*b* is represented as a Euclidean space with the following three quantities plotted along axes at right angles: L^* representing lightness, a^* representing the red/green axis, and b^* representing the yellow/blue axis. The formulas for 1976 CIE L*a*b* follow:

$$\begin{aligned}
 L^* &= 116(Y/Y_n)^{1/3} - 16 && \text{for } Y/Y_n > 0.008856 \\
 L^* &= 903.3(Y/Y_n) && \text{for } Y/Y_n \leq 0.008856 \quad * \text{see note below.} \\
 a^* &= 500[(X/X_n)^{1/3} - (Y/Y_n)^{1/3}] \\
 b^* &= 200[(Y/Y_n)^{1/3} - (Z/Z_n)^{1/3}].
 \end{aligned}$$

where X_n , Y_n , and Z_n are the CIE X, Y, and Z tristimulus values of an *appropriate* reference white. Also, if any of the ratios X/X_n , Y/Y_n , or Z/Z_n is equal to or less than 0.008856, it is replaced in the formulas with

$$7.787F + 16/116,$$

where F is X/X_n , Y/Y_n , or Z/Z_n , as appropriate (note: these low-light conditions are of no relevance for most document-imaging applications). Tiff is defined such that each quantity be encoded with 8 bits. This provides 256 levels of L^* lightness; 256 levels (+/- 127) of a^* , and 256 levels (+/- 127) of b^* . Dividing the 0-100 range of L^* into 256 levels provides lightness steps that are less than half the size of a "just noticeable difference". This eliminates banding, even under conditions of substantial tonal manipulation. Limiting the theoretically unbounded a^* and b^* ranges to +/- 127 allows encoding in 8 bits without eliminating any but the most saturated self-luminous colors. It is anticipated that the rare specialized applications requiring support of these extreme cases would be unlikely to use CIELAB anyway. All object colors, in fact all colors within the theoretical MacAdam limits, fall within the +/- 127 a^*/b^* range.

The TIFF CIELAB Fields

Photometric Interpretation

Tag = 262 (106.H)

Type = SHORT

N = 1

8 = 1976 CIE $L^*a^*b^*$

Usage of other Fields.

BitsPerSample: 8

SamplesPerPixel - ExtraSamples: 3 for $L^*a^*b^*$, 1 implies L^* only, for monochrome data.

Compression: same as other multi-bit formats. JPEG compression applies.

PlanarConfiguration: both chunky and planar data could be supported.

WhitePoint: does not apply

PrimaryChromaticities: does not apply.

TransferFunction: does not apply

Alpha Channel information will follow the lead of other data types.

The reference white for this data type is the *perfect reflecting diffuser* (100% diffuse reflectance at all visible wavelengths). The L^* range is from 0 (perfect absorbing black) to 100 (perfect reflecting diffuse white). The a^* and b^* ranges will be represented as signed 8 bit values having the range -127 to +127.

Converting between RGB and CIELAB, a Caveat

The above CIELAB formulae are derived from CIE XYZ. Converting from CIELAB to RGB requires an additional set of formulae for converting between RGB and XYZ. For standard NTSC primaries these are:

0.6070	0.1740	0.2000		R	X
0.2990	0.5870	0.1140	*	G	= Y
0.0000	0.0660	1.1110		B	Z

Generally, D65 illumination is used and a perfect reflecting diffuser is used for the reference white.

Since CIELAB is not a directly displayable format, some conversion to RGB will be required. While look-up table accelerated CIELAB to RGB conversion is certainly possible and fast, TIFF writers may choose to include a low resolution RGB subfile as an integral part of TIFF CIELAB.

Color Difference Measurements in CIELAB

The differences between two colors in L^* , a^* , and b^* are denoted by DL^* , Da^* , and Db^* , respectively, with the total (3-dimensional) color difference represented as:

$$E^*_{ab} = [(L^*)^2 + (a^*)^2 + (b^*)^2]^{1/2}.$$

This color difference can also be expressed in terms of L^* , C^* , and a measure of hue. In this case, h_{ab} is *not* used because it is an angular measure and cannot be combined with L^* and C^* directly. A linear-distance form of hue is used instead:

*CIE 1976 a,b hue-difference, H^*_{ab}*

$$H^*_{ab} = [(E^*)^2 - (L^*)^2 - (C^*)^2]^{1/2}.$$

where DC^* is the chroma difference between the two colors. The total color difference expression using this hue-difference is:

$$E^*_{ab} = [(L^*)^2 + (H^*)^2 + (b^*)^2]^{1/2}.$$

It is important to remember that color difference is 3-dimensional: much more can be learned from a $DL^*a^*b^*$ triplet than from a single DE value. The $DL^*C^*H^*$ form is often the most useful since it gives the error information in a form that has more familiar perception correlates. Caution is in order, however, when using DH^* for large hue differences since it is a straight-line approximation of a curved hue distance.

The Merits of CIELAB

Colorimetric.

First and foremost, CIELAB is colorimetric. It is traceable to the internationally-recognized standard CIE 1931 Standard Observer. This insures that it encodes color in a manner that is accurately modeled after the human vision system. Colors seen as matching are encoded identically, and colors seen as not matching are encoded differently. CIELAB provides an unambiguous definition of color without the necessity of additional information such as with RGB (primary chromaticities, white point, and gamma curves).

Device Independent.

Unlike RGB spaces which associate closely with physical phosphor colors, CIELAB contains no device association. CIELAB is not tailored for one device or device type at the expense of all others.

Full Color Gamut.

Any one image or imaging device usually encounters a very limited subset of the entire range of humanly-perceptible color. Collectively, however, these images and devices span a much larger gamut of color. A truly versatile exchange color space should encompass all of these colors, ideally providing support for all visible color. RGB, PhotoYCC, YCbCr, and other display spaces suffer from gamut limitations that exclude significant regions of easily printable colors. CIELAB is defined for all visible color.

Efficiency

A good exchange space will maximize accuracy of translations between itself and other spaces. It will represent colors compactly for a given accuracy. These attributes are provided through visual uniformity. One of the greatest disadvantages of the classic CIE system (and RGB systems as well) is that colors within it are not equally spaced visually. Encoding full-color images in a linear-intensity space, such as the typical RGB space or, especially, the XYZ space, requires a very large range (greater than 8-bits/primary) to eliminate banding artifacts. Adopting a *non-linear* RGB space improves the efficiency but not nearly to the extent as with a perceptually uniform space where these problems are nearly eliminated. A uniform space is also more efficiently compressed (see below).

Public Domain / Single Standard

CIELAB maintains no preferential attachments to any private organization. Its existence as a single standard leaves no room for ambiguity. Since 1976, CIELAB has continually gained popularity as a widely-accepted and heavily-used standard.

Luminance/Chrominance Separation.

The advantages for image size compression made possible by having a separate lightness or luminance channel are immense. Many such spaces exist. The degree to which the luminance information is fully-isolated into a single channel is an important consideration. Recent studies (Kasson and Plouffe of IBM) support CIELAB as a leading candidate placing it above CIELUV, YIQ, YUV, YCC, and XYZ.

Other advantages support a separate lightness or luminance channel. Tone and contrast editing and detail enhancement are most easily accomplished with such a channel. Conversion to a black and white representation is also easiest with this type of space.

When the chrominance channels are encoded as opponents as with CIELAB, there are other compression, image manipulation, and white point handling advantages.

Compressibility (Data).

Opponent spaces such as CIELAB are inherently more compressible than tristimulus spaces such as RGB. The chroma content of an image can be compressed to a greater extent, without objectionable loss, than can the lightness content. The opponent arrangement of CIELAB allows for spatial subsampling and efficient compression using JPEG.

Compressibility (Gamut).

Adjusting the color range of an image to match the capabilities of the intended output device is a critical function within computational color reproduction. Luminance/chrominance separation, especially when provided in a polar form, is desirable for facilitating gamut compression. Accurate gamut compression in a tri-linear color space is difficult.

CIELAB has a polar form (*metric hue angle*, and *metric chroma*, described below) that serves compression needs fairly well. Because CIELAB is not perfectly uniform, problems can arise when compressing along constant hue lines. Noticeable hue errors are sometimes introduced. This problem is no less severe with other contending color spaces.

This polar form also provides advantages for local color editing of images. The polar form is not proposed as part of the TIFF addition.

Getting the Most from CIELAB

Image Editors

The advantages of image editing within a perceptually uniform polar color space are tremendous. A detailed description of these advantages is beyond the scope of this section. As previously mentioned, many common tonal manipulation tasks are most efficiently performed when only a single channel is affected. Edge enhancement, contrast adjustment, and general tone-curve manipulation all ideally affect only the lightness component of an image.

A perceptual polar space works excellently for specifying a color range for masking purposes. For example, a red shirt can be quickly changed to a green shirt without drawing an outline mask. The operation can be performed with a loosely, quickly-drawn mask region combined with a hue (and perhaps chroma) range that encompasses the shirt's colors. The hue component of the shirt can then be adjusted, leaving the lightness and chroma detail in place.

Color cast adjustment is easily realized by shifting either or both of the chroma channels over the entire image or blending them over the region of interest.

Converting from CIELAB to a device specific space

For fast conversion to an RGB display, CIELAB can be decoded using 3x3 matrixing followed by gamma correction. The computational complexity required

for accurate CRT display is the same with CIELAB as with extended luminance-chrominance spaces.

Converting CIELAB for accurate printing on CMYK devices requires computational complexity no greater than with *accurate* conversion from any other colorimetric space. Gamut compression becomes one of the more significant tasks for any such conversion.

Part 3: Appendices

Part 3 contains additional information that is not part of the TIFF specification, but may be of use to developers.

Appendix A: TIFF Tags Sorted by Number

TagName	Decimal	Hex	Type	Number of values
NewSubfileType	254	FE	LONG	1
SubfileType	255	FF	SHORT	1
ImageWidth	256	100	SHORT or LONG	1
ImageLength	257	101	SHORT or LONG	1
BitsPerSample	258	102	SHORT	SamplesPerPixel
Compression	259	103	SHORT	1
Uncompressed	1			
CCITT 1D	2			
Group 3 Fax	3			
Group 4 Fax	4			
LZW	5			
JPEG	6			
PackBits	32773			
PhotometricInterpretation	262	106	SHORT	1
WhiteIsZero	0			
BlackIsZero	1			
RGB	2			
RGB Palette	3			
Transparency mask	4			
CMYK	5			
YCbCr	6			
CIELab	8			
Threshholding	263	107	SHORT	1
CellWidth	264	108	SHORT	1
CellLength	265	109	SHORT	1
FillOrder	266	10A	SHORT	1
DocumentName	269	10D	ASCII	
ImageDescription	270	10E	ASCII	
Make	271	10F	ASCII	
Model	272	110	ASCII	
StripOffsets	273	111	SHORT or LONG	StripsPerImage
Orientation	274	112	SHORT	1
SamplesPerPixel	277	115	SHORT	1
RowsPerStrip	278	116	SHORT or LONG	1
StripByteCounts	279	117	LONG or SHORT	StripsPerImage
MinSampleValue	280	118	SHORT	SamplesPerPixel
MaxSampleValue	281	119	SHORT	SamplesPerPixel
XResolution	282	11A	RATIONAL	1
YResolution	283	11B	RATIONAL	1
PlanarConfiguration	284	11C	SHORT	1
PageName	285	11D	ASCII	
XPosition	286	11E	RATIONAL	
YPosition	287	11F	RATIONAL	
FreeOffsets	288	120	LONG	
FreeByteCounts	289	121	LONG	
GrayResponseUnit	290	122	SHORT	1

GrayResponseCurve	291	123	SHORT	$2^{**}\text{BitsPerSample}$
T4Options	292	124	LONG	1
T6Options	293	125	LONG	1
ResolutionUnit	296	128	SHORT	1
PageNumber	297	129	SHORT	2
TransferFunction	301	12D	SHORT	{ 1 or SamplesPerPixel}* $2^{**}\text{ BitsPerSample}$
Software	305	131	ASCII	
DateTime	306	132	ASCII	20
Artist	315	13B	ASCII	
HostComputer	316	13C	ASCII	
Predictor	317	13D	SHORT	1
WhitePoint	318	13E	RATIONAL	2
PrimaryChromaticities	319	13F	RATIONAL	6
ColorMap	320	140	SHORT	$3 * (2^{**}\text{BitsPerSample})$
HalftoneHints	321	141	SHORT	2
TileWidth	322	142	SHORT or LONG	1
TileLength	323	143	SHORT or LONG	1
TileOffsets	324	144	LONG	TilesPerImage
TileByteCounts	325	145	SHORT or LONG	TilesPerImage
InkSet	332	14C	SHORT	1
InkNames	333	14D	ASCII	total number of charac- ters in all ink name strings, including zeros
NumberOfInks	334	14E	SHORT	1
DotRange	336	150	BYTE or SHORT	2, or 2^* NumberOfInks
TargetPrinter	337	151	ASCII	any
ExtraSamples	338	152	BYTE	number of extra compo- nents per pixel
SampleFormat	339	153	SHORT	SamplesPerPixel
SMinSampleValue	340	154	Any	SamplesPerPixel
SMaxSampleValue	341	155	Any	SamplesPerPixel
TransferRange	342	156	SHORT	6
JPEGProc	512	200	SHORT	1
JPEGInterchangeFormat	513	201	LONG	1
JPEGInterchangeFormatLength	514	202	LONG	1
JPEGRestartInterval	515	203	SHORT	1
JPEGLosslessPredictors	517	205	SHORT	SamplesPerPixel
JPEGPointTransforms	518	206	SHORT	SamplesPerPixel
JPEGQTables	519	207	LONG	SamplesPerPixel
JPEGDCTables	520	208	LONG	SamplesPerPixel
JPEGACTables	521	209	LONG	SamplesPerPixel
YCbCrCoefficients	529	211	RATIONAL	3
YCbCrSubSampling	530	212	SHORT	2
YCbCrPositioning	531	213	SHORT	1
ReferenceBlackWhite	532	214	LONG	$2^*\text{SamplesPerPixel}$
Copyright	33432	8298	ASCII	Any

Appendix B: Operating System Considerations

Extensions and Filetypes

The recommended MS-DOS, UNIX, and OS/2 file extension for TIFF files is “.TIF”.

On an Apple Macintosh computer, the recommended Filetype is “TIFF”. It is a good idea to also name TIFF files with a “.TIF” extension so that they can easily imported if transferred to a different operating system.

Index

Symbols

42 13

A

Aldus Developers Desk 8
 alpha data 31
 associated 69
 ANSI IT8 71
 Appendices 116
 AppleLink 8
 Artist 28
 ASCII 15

B

Baseline TIFF 11
 big-endian 13
 BitsPerSample 22, 29
 BlackIsZero 17, 37
 BYTE data type 15

C

CCITT 17, 30, 49
 CellLength 29
 CellWidth 29
 chunky format 38
 CIELAB images 110
 clarifications 6
 Class B 21
 Class G 22
 Class P 23
 Class R 25
 Classes 7
 CMYK Images 66
 ColorMap 23, 29
 ColorResponseCurves. *See*
 TransferFunction
 compatibility 6
 compliance 12
 component 28
 compositing. *See* alpha data:
 associated

compression 17, 30
 CCITT 49
 JPEG 95
 LZW 57
 Modified Huffman 43
 PackBits 42
 CompuServe 8
 Copyright 31
 Count 14, 15, 16

D

DateTime 31
 default values 28
 Differencing Predictor 64
 DocumentName 55
 DotRange 71
 DOUBLE 16
 Duff, Tom 79

E

ExtraSamples 31, 77

F

Facsimile 49
 file extension 119
 filetype 119
 FillOrder 32
 FLOAT 16
 FreeByteCounts 33
 FreeOffsets 33

G

GrayResponseCurve 33, 73, 85
 GrayResponseUnit 33
 Group 3 17, 30
 Group3Options 51
 Group4Options 52

H

HalftoneHints 72
 Hexadecimal 12
 high fidelity color 69
 HostComputer 34

I

IFD. *See* image file directory
 II 13
 image 28
 image file directory 13, 14
 image file header 13
 ImageDescription 34
 ImageLength 18, 27, 34
 ImageWidth 18, 27, 34
 InkNames 70
 InkSet 70

J

JPEG compression 95
 baseline 95
 discrete cosine trans-
 form 95
 entropy coding 98
 lossless processes 100
 quantization 97
 JPEGACTables 107
 JPEGDCTables 107
 JPEGInterchangeFormat 105
 JPEGInterchangeFormatLength 105
 JPEGLosslessPredictors 106
 JPEGPointTransforms 106
 JPEGProc 104
 JPEGQTables 107
 JPEGRestartInterval 105

K

no entries

L

little-endian 13
 LONG data type 15
 LZW compression 57

M

Make 35
 matting. *See* alpha data: associ-
 ated
 MaxComponentValue 35
 MaxSampleValue. *See*
 MaxComponentValue
 MinComponentValue 35

MinSampleValue. *See*
 MinComponentValue
 MM 13
 Model 35
 Modified Huffman compression 17, 30, 43
 multi-page TIFF files 36
 multiple strips 39

N

NewSubfileType 36
 NumberOfInks 70

O

Offset 15
 Orientation 36

P

PackBits compression 42
 PageName 55
 PageNumber 55
 palette color 23, 29, 37
 PhotometricInterpretation 17, 32, 37
 pixel 28
 planar format 38
 PlanarConfiguration 38
 Porter, Thomas 79
 Predictor 64
 PrimaryChromaticities 83
 private tags 8
 proposals
 submitting 9

Q

no entries

R

RATIONAL data type 15
 reduced resolution 36
 ReferenceBlackWhite 86
 ResolutionUnit 18, 27, 38
 revision notes 4
 RGB images 37
 row interleave 38
 RowsPerStrip 19, 27, 39, 68

S

sample. *See* component
 SampleFormat 80
 SamplesPerPixel 39

SBYTE 16
 separated images 66
 SHORT data type 15
 SLONG 16
 Software 39
 SRATIONAL 16
 SSHORT 16
 StripByteCounts 19, 27, 40
 StripOffsets 19, 27, 40
 StripsPerImage 39
 subfile 16
 SubfileType 40. *See also*
 NewSubfileType

T

T4Options 51
 T6Options 52
 tag 14
 TargetPrinter 71
 Thresholding 41
 TIFF
 administration 8
 Baseline 11
 Class P 23
 Class R 24
 Classes 17
 consulting 8
 extensions 48
 history 4
 other extensions 9
 sample Files 20
 scope 4
 structure 13
 tags - sorted 117
 TIFF Advisory Committee 9
 TileByteCounts 68
 TileLength 67
 TileOffsets 68
 Tiles 66
 TilesPerImage 67, 68
 TileWidth 67
 TransferFunction 84
 TransferRange 86
 transparency mask 36, 37
 type of a field 14

U

UNDEFINED 16

V

no entries

W

WhitelsZero 17, 37
 WhitePoint 83

X

XPosition 55
 XResolution 19, 27, 41

Y

YCbCr images 87, 89
 YCbCrCoefficients 91
 YCbCrPositioning 92
 YCbCrSubSampling 91
 YPosition 56
 YResolution 19, 41

Z

no entries

COMP630 WAV File Format Description



WAV files are probably the simplest of the common formats for storing audio samples. Unlike MPEG and other compressed formats, WAVs store samples "in the raw" where no pre-processing is required other than formatting of the data.

The following information was derived from several sources including some on the internet which no longer exist. Being somewhat of a proprietary Microsoft format there are some elements here which were empirically determined and so some details may remain somewhat sketchy. From what I've heard, the best source for information is the *File Formats Handbook* by Gunter Born (1995, ITP Boston)

The WAV file itself consists of three "chunks" of information: The RIFF chunk which identifies the file as a WAV file, The FORMAT chunk which identifies parameters such as sample rate and the DATA chunk which contains the actual data (samples).

Each Chunk breaks down as follows:

RIFF Chunk (12 bytes in length total)

Byte Number	
0 - 3	"RIFF" (ASCII Characters)
4 - 7	Total Length Of Package To Follow (Binary, little endian)
8 - 11	"WAVE" (ASCII Characters)

FORMAT Chunk (24 bytes in length total)

Byte Number	
0 - 3	"fmt_" (ASCII Characters)
4 - 7	Length Of FORMAT Chunk (Binary, always 0x10)
8 - 9	Always 0x01
10 - 11	Channel Numbers (Always 0x01=Mono, 0x02=Stereo)
12 - 15	Sample Rate (Binary, in Hz)
16 - 19	Bytes Per Second
20 - 21	Bytes Per Sample: 1=8 bit Mono, 2=8 bit Stereo or 16 bit Mono, 4=16 bit Stereo
22 - 23	Bits Per Sample

DATA Chunk

Byte Number	
0 - 3	"data" (ASCII Characters)
4 - 7	Length Of Data To Follow
8 - end	Data (Samples)

The easiest approach to this file format might be to look at an actual WAV file to see how data is stored. In this case, we examine DING.WAV which is standard with all Windows packages. DING.WAV is an 8-bit, mono, 22.050 KHz WAV file of 11,598 bytes in length. Lets begin by looking at the header of the file (using DEBUG).

```

246E:0100  52 49 46 46 46 2D 00 00-57 41 56 45 66 6D 74 20  RIFFF-..WAVEfmt
246E:0110  10 00 00 00 01 00 01 00-22 56 00 00 22 56 00 00  ..... "V.."V..
246E:0120  01 00 08 00 64 61 74 61-22 2D 00 00 80 80 80 80  ....data"-.....
246E:0130  80 80 80 80 80 80 80 80-80 80 80 80 80 80 80 80  .....
246E:0140  80 80 80 80 80 80 80 80-80 80 80 80 80 80 80 80  .....

```

As expected, the file begins with the ASCII characters "RIFF" identifying it as a WAV file. The next four bytes tell us the length is 0x2D46 bytes (11590 bytes in decimal) which is the length of the entire file minus the 8 bytes for the "RIFF" and length (11598 - 11590 = 8 bytes).

The ASCII characters for "WAVE" and "fmt " follow. Next (line 2 above) we find the value 0x00000010 in the first 4 bytes (length of format chunk: always constant at 0x10). The next four bytes are 0x0001 (Always) and 0x0001 (A mono WAV, one channel used).

Since this is a 8-bit WAV, the sample rate and the bytes/second are the same at 0x00005622 or 22,050 in decimal. For a 16-bit stereo WAV the bytes/sec would be 4 times the sample rate. The next 2 bytes show the number of bytes per sample to be 0x0001 (8-bit mono) and the number of bits per sample to be 0x0008.

Finally, the ASCII characters for "data" appear followed by 0x00002D22 (11,554 decimal) which is the number of bytes of data to follow (actual samples). The data is a value from 0x00 to 0xFF. In the example above 0x80 would represent "0" or silence on the output since the DAC used to playback samples is a bipolar device (i.e. a value of 0x00 would output a negative voltage and a value of 0xFF would output a positive voltage at the output of the DAC on the sound card).

Note that there are extension to the basic WAV format which may be supported in newer systems -- for example if you look at DING.WAV in Windows '95 you'll see some extra bytes added after the format chunk before the "data" area -- but the basic format remains the same.

As a final example consider the header for the following WAV file recorded at 44,100 samples per second in 16-bit stereo.

```

246E:0100  52 49 46 46 2C 48 00 00-57 41 56 45 66 6D 74 20  RIFF,H..WAVEfmt
246E:0110  10 00 00 00 01 00 02 00-44 AC 00 00 10 B1 02 00  .....D.....
246E:0120  04 00 10 00 64 61 74 61-00 48 00 00 00 00 00 00  ....data.H.....
246E:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  .....

```

Again we find all the expected structures. Note that the sample rate is 0xAC44 (44,100 as an unsigned int in decimal) and the bytes/second is 4 times that figure since this is a 16-bit WAV (* 2) and is stereo (again * 2). The Channel Numbers field is also found to be 0x02 here and the bits per sample is 0x10 (16 decimal).



[Back to the course Home Page](#)

Audio & Multimedia

MPEG

Audio Layer-3

- History
 - Quality
 - Details
- ▶ Fraunhofer AEMT

History

In 1987, the Fraunhofer IIS started to audio coding in the framework of the EU147, Digital Audio Broadcasting (cooperation with the University of Erlangen-Nürnberg), the Fraunhofer IIS finally developed an algorithm that is standardized as ISO-MPEG Layer-3 (IS 11172-3 and IS 13818-3)

Without data reduction, digital audio consists of 16 bit samples recorded at a rate that is more than twice the actual audio bandwidth (Compact Discs). So you end up with **Mbit** to represent just **one second of quality**. By using MPEG audio coding, you reduce down the original sound data from a **Mbit** without losing sound quality. Factors that still maintain a sound quality that is better than what you get by just reducing the resolution of your samples. Basically, by *perceptual coding* techniques added to the original sound waves by the human ear.

Using MPEG audio, one may achieve a significant reduction of

1:4	by Layer 1 (corresponds to a mono signal),
1:6...1:8	by Layer 2 (corresponds to a stereo signal),
1:10...1:12	by Layer 3 (corresponds to a stereo signal),

still maintaining the original CD source quality.

By exploiting stereo effects and by limiting the audio bandwidth, the coding schemes may achieve a significant reduction of sound quality at even lower bitrates. The most powerful member of the MPEG family is MPEG-1 Layer 3. For a given sound quality level, it requires a lower bitrate - or for a given bitrate, it achieves a higher sound quality.

quality.

Sound Quality

Some typical performance data of M

sound quality	bandwidth	mode	b
telephone sound	2.5 kHz	mono	8
better than short wave	4.5 kHz	mono	1
better than AM radio	7.5 kHz	mono	3
similar to FM radio	11 kHz	stereo	56..
near-CD	15 kHz	stereo	9
CD	>15 kHz	stereo	112.

*) Fraunhofer IIS uses a non-ISO ex Layer-3 for enhanced performance (

In all international listening tests, MF impressively proved its superior performance at the original sound quality at a data rate (around 64 kbit/s per audio channel). It can tolerate a limited bandwidth of around 11 kHz, which provides a reasonable sound quality for stereo signals even at a reduction of 1:24.

For the use of low bit-rate audio coding in broadcast applications at bitrates of 64 kbit/s per channel, the ITU-R recommends MP3 (ITU-R doc. BS.1115)

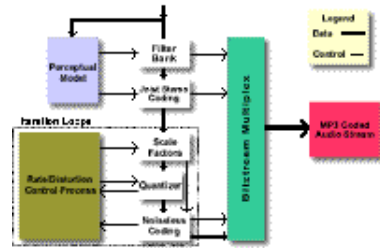


Fig.1: MP3 encoder flowchart

Details

Filter bank

The filter bank used in MPEG Layer-3 consists of a polyphase filter bank. The Modified Discrete Cosine Transform (MDCT) form was chosen for reasons of compactness over predecessors, Layer-1 and Layer-2.

Perceptual Model

The perceptual model mainly determines the quantization step size for a given encoder implementation. It uses the filter bank or combines the calculation (for the masking calculations) and the filter bank. The output of the perceptual model is the masking threshold or the allowed noise level. If the quantization noise can be made below the masking threshold, then the compressed signal is indistinguishable from the original signal.

Joint Stereo

Joint stereo coding takes advantage of the correlation between channels of a stereo channel pair containing common information. These stereophonic irredundancies are exploited to reduce the amount of data. Joint stereo is used in cases where only low bit-rate is available but stereo signals are desired.

Quantization and Coding

A system of two nested iteration loops is used for quantization and coding in the MP3 encoder.

Quantization is done via a power-law quantization. In this way, larger values are automatically quantized with higher accuracy and some noise shaping is used in the quantization process.

The quantized values are coded by Huffman coding, a specific method for entropy coding, which is lossless. This is called noiseless coding. The quantization noise is added to the audio signal.

The process to find the optimum gain for a given block, bit-rate and output format is done in the MP3 encoder.

model is usually done by two nested analysis-by-synthesis way:

- **Inner iteration loop (rate loop)**

The Huffman code tables assign to (more frequent) smaller quantization step sizes. If the number of bits resulting from the Huffman coding exceeds the number of bits available for a block of data, this can be corrected by increasing the quantization step size. This leads to a global gain to result in a larger quantization step size, leading to smaller quantized values. This process is repeated with different quantization step sizes until the resulting bit demand for Huffman coding is small enough. The loop is called rate control loop because it modifies the overall coder rate to be small enough.

- **Outer iteration loop (noise control loop)**

To shape the quantization noise spectrum, a noise masking threshold is used. The scalefactors are adjusted to meet the noise masking threshold. The systems use a scalefactor band. The systems use a scalefactor of 1.0 for each band. If the actual noise in a given band is found to exceed the masking threshold (allowed noise) as specified in a perceptual model, the scalefactors are adjusted to reduce the quantization noise. This process is repeated until achieving a smaller quantization step size. This leads to a larger number of quantization steps, which increases the bitrate. The rate adjustment loop is used to adjust the bitrate every time new scalefactors are calculated. The rate loop is nested within the noise control loop. The outer (noise control) loop is used to adjust the scalefactors until the actual noise (computed from the quantized values minus the original spectral values minus the quantization noise) is below the masking threshold. The scalefactor band (i.e. critical band) is used to determine the scalefactor for each band.



Audio & Multimedia

MPEG

Audio Layer-3

- History
 - Quality
 - Details
- ▶ Fraunhofer AEMT

History

In 1987, the Fraunhofer IIS started to audio coding in the framework of the EU147, Digital Audio Broadcasting (cooperation with the University of Erlangen-Nürnberg), the Fraunhofer IIS finally developed an algorithm that is standardized as ISO-MPEG-1 Layer-3 (IS 11172-3 and IS 13818-3)

Without data reduction, digital audio consists of 16 bit samples recorded at a rate that is more than twice the actual audio bandwidth (as used in Compact Discs). So you end up with **Mbit** to represent just **one second of quality**. By using MPEG audio coding, you reduce down the original sound data from a **Mbit** without losing sound quality. Factors that still maintain a sound quality that is better than what you get by just reducing the resolution of your samples. Basically, by *perceptual coding* techniques added to the original sound waves by the human ear.

Using MPEG audio, one may achieve a 4:1 reduction of

1:4	by Layer 1 (corresponds to a mono signal),
1:6...1:8	by Layer 2 (corresponds to a stereo signal),
1:10...1:12	by Layer 3 (corresponds to a stereo signal),

still maintaining the original CD source quality.

By exploiting stereo effects and by limiting the audio bandwidth, the coding schemes may achieve a 4:1 reduction of sound quality at even lower bitrates. The most powerful member of the MPEG family is MPEG-1 Layer 3. For a given sound quality level, it requires a 4:1 reduction of bitrate - or for a given bitrate, it achieves a 4:1 reduction of sound quality.

quality.

Sound Quality

Some typical performance data of M

sound quality	bandwidth	mode	b
telephone sound	2.5 kHz	mono	8
better than short wave	4.5 kHz	mono	1
better than AM radio	7.5 kHz	mono	3
similar to FM radio	11 kHz	stereo	56..
near-CD	15 kHz	stereo	9
CD	>15 kHz	stereo	112.

*) Fraunhofer IIS uses a non-ISO ex Layer-3 for enhanced performance (

In all international listening tests, MF impressively proved its superior performance at the original sound quality at a data rate (around 64 kbit/s per audio channel). It can tolerate a limited bandwidth of around 11 kHz, which provides a reasonable sound quality for stereo signals even at a reduction of 1:24.

For the use of low bit-rate audio coding in broadcast applications at bitrates of 64 kbit/s per channel, the ITU-R recommends MP3 (ITU-R doc. BS.1115)

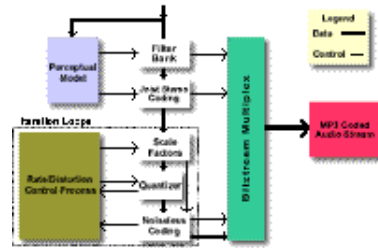


Fig.1: MP3 encoder flowchart

Details

Filter bank

The filter bank used in MPEG Layer-3 which consists of a polyphase filter bank. The Modified Discrete Cosine Transform (MDCT) form was chosen for reasons of compactness, Layer-1 and Layer-2.

Perceptual Model

The perceptual model mainly determines the masking threshold for a given encoder implementation. It uses the filter bank or combines the calculation (for the masking calculations) and the filter bank. The output of the perceptual model is the masking threshold or the allowed partition. If the quantization noise can be made below the masking threshold, then the compressed signal is indistinguishable from the original signal.

Joint Stereo

Joint stereo coding takes advantage of the common information in two channels of a stereo channel pair. These stereophonic irredundancies are exploited to reduce the amount of data. Joint stereo is used in cases where only low bit-rate is available but stereo signals are desired.

Quantization and Coding

A system of two nested iteration loops is used for quantization and coding in the MDCT domain.

Quantization is done via a power-law quantizer. In this way, larger values are automatically coded with higher accuracy and some noise shaping is applied during the quantization process.

The quantized values are coded by Huffman coding, a specific method for entropy coding, which is lossless. This is called noiseless coding. The quantization noise is added to the audio signal.

The process to find the optimum gain for a given block, bit-rate and output format is done by a search algorithm.

model is usually done by two nested analysis-by-synthesis way:

- **Inner iteration loop (rate loop)**

The Huffman code tables assign to (more frequent) smaller quantization step size. If the number of bits resulting from the Huffman code exceeds the number of bits available for a block of data, this can be corrected by increasing the quantization step size. This leads to a global gain to result in a larger quantization step size leading to smaller quantized values. This process is repeated with different quantization step sizes until the resulting bit demand for Huffman coding is small enough. The loop is called rate control loop because it modifies the overall coder rate to be small enough.

- **Outer iteration loop (noise control loop)**

To shape the quantization noise spectrum, a noise masking threshold is used. The scalefactors are adjusted in each scalefactor band. The systems use a scalefactor of 1.0 for each band. If the actual noise in a given band is found to exceed the masking threshold (allowed noise) as specified in a perceptual model, the scalefactors are adjusted to reduce the quantization noise, achieving a smaller quantization step size. This leads to a larger number of quantization steps and a larger number of bits. In the rate adjustment loop, every time new scalefactors are calculated, the rate loop is nested within the noise control loop. The outer (noise control) loop is repeated until the actual noise (computed from the original spectral values minus the quantized values) is below the masking threshold in each scalefactor band (i.e. critical band).



This is a brief and informal document targeted to those who want to deal with the MPEG format. If you are one of them, you probably already know what is MPEG audio. If not, jump to <http://www.mp3.com/> or <http://www.layer3.org/> where you will find more details and also more links. This document does not cover compression and decompression algorithm.

NOTE: You cannot just search the Internet and find the MPEG audio specs. It is copyrighted and you will have to pay quite a bit to get the Paper. That's why I made this. Information I got is gathered from the Internet, and mostly originate from program sources I found available for free. Despite my intention to always specify the information sources, I am not able to do it this time. Sorry, I did not maintain the list. :-)

These are not a decoding specs, it just informs you how to read the MPEG headers and the MPEG TAG. MPEG Version 1, 2 and 2.5 and Layer I, II and III are supported, the MP3 TAG (ID3v1 and ID3v1.1) also. Those of you who use Delphi may find MPGTools Delphi unit (freeware source) useful, it is where I implemented this stuff.

I do not claim information presented in this document is accurate. At first I just gathered it from different sources. It was not an easy task but I needed it. Later, I received lots of comments as feedback when I published this document. I think this last release is highly accurate due to comments and corrections I received.

This document is last updated on December 22, 1999.

MPEG Audio Compression Basics

This is one of many methods to compress audio in digital form trying to consume as little space as possible but keep audio quality as good as possible. MPEG compression showed up as one of the best achievements in this area.

This is a lossy compression, which means, you will certainly lose some audio information when you use this compression methods. But, this loss can hardly be noticed because the compression method tries to control it. By using several quite complicated and demanding mathematical algorithms it will only lose those parts of sound that are hard to be heard even in the original form. This leaves more space for information that is important. This way you can compress audio up to 12 times (you may choose compression ratio) which is really significant. Due to its quality MPEG audio became very popular.

MPEG standards MPEG-1, MPEG-2 and MPEG-4 are known but this document covers first two of them. There is an unofficial MPEG-2.5 which is rarely used. It is also covered.

MPEG-1 audio (described in ISO/IEC 11172-3) describes three Layers of audio coding with the following properties:

- one or two audio channels
- sample rate 32kHz, 44.1kHz or 48kHz.
- bit rates from 32kbps up to 448kbps

Each layer has its merits.

MPEG-2 audio (described in ISO/IEC 13818-3) has two extensions to MPEG-1, usually referred as

MPEG-2/LSF and MPEG-2/Multichannel.

MPEG-2/LSF has the following properties:

- one or two audio channels
- sample rates half those of MPEG-1
- bit rates from 8 kbps up to 256kbps.

MPEG-2/Multichannel has the following properties:

- up to 5 full range audio channels and an LFE-channel (Low Frequency Enhancement <> subwoofer!)
- sample rates the same as those of MPEG-1
- highest possible bitrate goes up to about 1Mbps for 5.1

MPEG Audio Frame Header

An MPEG audio file is built up from smaller parts called frames. Generally, frames are independent items. Each frame has its own header and audio informations. There is no file header. Therefore, you can cut any part of MPEG file and play it correctly (this should be done on frame boundaries but most applications will handle incorrect headers). For Layer III, this is not 100% correct. Due to internal data organization in MPEG version 1 Layer III files, frames are often dependent of each other and they cannot be cut off just like that.

When you want to read info about an MPEG file, it is usually enough to find the first frame, read its header and assume that the other frames are the same This may not be always the case. Variable bitrate MPEG files may use so called bitrate switching, which means that bitrate changes according to the content of each frame. This way lower bitrates may be used in frames where it will not reduce sound quality. This allows making better compression while keeping high quality of sound.

The frame header is constituted by the very first four bytes (32bits) in a frame. The first eleven bits (or first twelve bits, see below about frame sync) of a frame header are always set and they are called "frame sync". Therefore, you can search through the file for the first occurrence of frame sync (meaning that you have to find a byte with a value of 255, and followed by a byte with its three (or four) most significant bits set). Then you read the whole header and check if the values are correct. You will see in the following table the exact meaning of each bit in the header, and which values may be checked for validity. Each value that is specified as reserved, invalid, bad, or not allowed should indicate an invalid header. Remember, this is not enough, frame sync can be easily (and very frequently) found in any binary file. Also it is likely that MPEG file contains garbage on it's beginning which also may contain false sync. Thus, you have to check two or more frames in a row to assure you are really dealing with MPEG audio file.

Frames may have a CRC check. The CRC is 16 bits long and, if it exists, it follows the frame header. After the CRC comes the audio data. You may calculate the length of the frame and use it if you need to read other headers too or just want to calculate the CRC of the frame, to compare it with the one you read from the file. This is actually a very good method to check the MPEG header validity.

Here is "graphical" presentation of the header content. Characters from A to M are used to indicate different fields. In the table, you can see details about the content of each field.

AAAAAAAA AAABBCCD EEEFFGH I IJJKLM

Sign	Length (bits)	Position (bits)	Description
A	11	(31-21)	Frame sync (all bits set)
B	2	(20,19)	MPEG Audio version ID 00 - MPEG Version 2.5 01 - reserved 10 - MPEG Version 2 (ISO/IEC 13818-3) 11 - MPEG Version 1 (ISO/IEC 11172-3)
C	2	(18,17)	Layer description 00 - reserved 01 - Layer III 10 - Layer II 11 - Layer I
D	1	(16)	Protection bit 0 - Protected by CRC (16bit crc follows header) 1 - Not protected
E	4	(15,12)	Bitrate index

Note: MPEG Version 2.5 is not official standard. Bit No 20 in frame header is used to indicate version 2.5. Applications that do not support this MPEG version expect this bit always to be set, meaning that frame sync (A) is twelve bits long, not eleven as stated here. Accordingly, B is one bit long (represents only bit No 19). I recommend using methodology presented here, since this allows you to distinguish all three versions and keep full compatibility.

bits	V1,L1	V1,L2	V1,L3	V2,L1	V2, L2 & L3
0000	free	free	free	free	free
0001	32	32	32	32	8
0010	64	48	40	48	16
0011	96	56	48	56	24
0100	128	64	56	64	32
0101	160	80	64	80	40
0110	192	96	80	96	48
0111	224	112	96	112	56
1000	256	128	112	128	64
1001	288	160	128	144	80
1010	320	192	160	160	96
1011	352	224	192	176	112
1100	384	256	224	192	128
1101	416	320	256	224	144
1110	448	384	320	256	160
1111	bad	bad	bad	bad	bad

NOTES: All values are in kbps

V1 - MPEG Version 1

V2 - MPEG Version 2 and Version 2.5

L1 - Layer I

L2 - Layer II

L3 - Layer III

"free" means free format. If the correct fixed bitrate (such files cannot use variable bitrate) is different than those presented in upper table it must be determined by the application. This may be implemented only for internal purposes since third party applications have no means to find out correct bitrate. However, this is not impossible to do but demands lot's of efforts.

"bad" means that this is not an allowed value

MPEG files may have variable bitrate (VBR). This means that bitrate in the file may change. I have learned about two used methods:

bitrate switching. Each frame may be created with different bitrate. It may be used in all layers. Layer III decoders must support this method. Layer I & II decoders may support it.

bit reservoir. Bitrate may be borrowed (within limits) from previous frames in order to provide more bits to demanding parts of the input signal. This causes, however, that the frames are no longer independent, which means you should not cut this files. This is supported only in Layer III.

More about VBR you may find on Xing Tech site

For Layer II there are some combinations of bitrate and mode which are not allowed. Here is a list of allowed combinations.

bitrate	allowed modes
free	all
32	single channel
48	single channel
56	single channel
64	all
80	single channel
96	all
112	all
128	all
160	all
192	all
224	stereo, intensity stereo, dual channel
256	stereo, intensity stereo, dual channel
320	stereo, intensity stereo, dual channel
384	stereo, intensity stereo, dual channel

bits	MPEG1	MPEG2	MPEG2.5
00	44100	22050	11025
01	48000	24000	12000
10	32000	16000	8000
11	reserv.	reserv.	reserv.

- G 1 (9) Padding bit
 0 - frame is not padded
 1 - frame is padded with one extra slot
 Padding is used to fit the bit rates exactly. For an example: 128k 44.1kHz layer II uses a lot of 418 bytes and some of 417 bytes long frames to get the exact 128k bitrate. For Layer I slot is 32 bits long, for Layer II and Layer III slot is 8 bits long.

How to calculate frame length

First, let's distinguish two terms frame size and frame length. Frame size is the number of samples contained in a frame. It is constant and always 384 samples for Layer I and 1152 samples for Layer II and Layer III. Frame length is length of a frame when compressed. It is calculated in slots. One slot is 4 bytes long for Layer I, and one byte long for Layer II and Layer III. When you are reading MPEG file you must calculate this to be able to find each consecutive frame. Remember, frame length may change from frame to frame due to padding or bitrate switching.

Read the BitRate, SampleRate and Padding of the frame header.

For Layer I files use this formula:

$$\text{FrameLengthInBytes} = (12 * \text{BitRate} / \text{SampleRate} + \text{Padding}) * 4$$

For Layer II & III files use this formula:

$$\text{FrameLengthInBytes} = 144 * \text{BitRate} / \text{SampleRate} + \text{Padding}$$

Example:

Layer III, BitRate=128000, SampleRate=441000, Padding=0

==> FrameSize=417 bytes

- H 1 (8) Private bit. It may be freely used for specific needs of an application, i.e. if it has to trigger some application specific events.
- I 2 (7,6) Channel Mode
 00 - Stereo
 01 - Joint stereo (Stereo)
 10 - Dual channel (Stereo)
 11 - Single channel (Mono)
- J 2 (5,4) Mode extension (Only if Joint stereo)

Mode extension is used to join informations that are of no use for stereo effect, thus reducing needed resources. These bits are dynamically determined by an encoder in Joint stereo mode.

Complete frequency range of MPEG file is divided in subbands There are 32 subbands. For Layer I & II these two bits determine frequency range (bands) where intensity stereo is applied. For Layer III these two bits determine which type of joint stereo is used (intensity stereo or m/s stereo). Frequency range is determined within decompression algorithm.

	Layer I and II	Layer III	
value	Layer I & II	Intensity stereo	MS stereo
00	bands 4 to 31	off	off
01	bands 8 to 31	on	off
10	bands 12 to 31	off	on
11	bands 16 to 31	on	on

- K 1 (3) Copyright
0 - Audio is not copyrighted
1 - Audio is copyrighted
- L 1 (2) Original
0 - Copy of original media
1 - Original media
- M 2 (1,0) Emphasis
00 - none
01 - 50/15 ms
10 - reserved
11 - CCIT J.17

MPEG Audio Tag ID3v1

The TAG is used to describe the MPEG Audio file. It contains information about artist, title, album, publishing year and genre. There is some extra space for comments. It is exactly 128 bytes long and is located at very end of the audio data. You can get it by reading the last 128 bytes of the MPEG audio file.

**AAABBBBB BBBB BBBB BBBB BBBB
 BCCCCCCC CCCCCCCC CCCCCCCC CCCCCC
 DDDDDDDD DDDDDDDD DDDDDDDD DDDDEEEE
 EFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFG**

Sign Length Position Description
 (bytes) (bytes)

- A 3 (0-2) Tag identification. Must contain 'TAG' if tag

exists and is correct.

B	30	(3-32)	Title
C	30	(33-62)	Artist
D	30	(63-92)	Album
E	4	(93-96)	Year
F	30	(97-126)	Comment
G	1	(127)	Genre

The specification asks for all fields to be padded with null character (ASCII 0). However, not all applications respect this (an example is WinAmp which pads fields with <space>, ASCII 32).

There is a small change proposed in **ID3v1.1** structure. The last byte of the Comment field may be used to specify the track number of a song in an album. It should contain a null character (ASCII 0) if the information is unknown.

Genre is a numeric field which may have one of the following values:

0	'Blues'	20	'Alternative'	40	'AlternRock'	60	'Top 40'
1	'Classic Rock'	21	'Ska'	41	'Bass'	61	'Christian Rap'
2	'Country'	22	'Death Metal'	42	'Soul'	62	'Pop/Funk'
3	'Dance'	23	'Pranks'	43	'Punk'	63	'Jungle'
4	'Disco'	24	'Soundtrack'	44	'Space'	64	'Native American'
5	'Funk'	25	'Euro-Techno'	45	'Meditative'	65	'Cabaret'
6	'Grunge'	26	'Ambient'	46	'Instrumental Pop'	66	'New Wave'
7	'Hip-Hop'	27	'Trip-Hop'	47	'Instrumental Rock'	67	'Psychadelic'
8	'Jazz'	28	'Vocal'	48	'Ethnic'	68	'Rave'
9	'Metal'	29	'Jazz+Funk'	49	'Gothic'	69	'Showtunes'
10	'New Age'	30	'Fusion'	50	'Darkwave'	70	'Trailer'
11	'Oldies'	31	'Trance'	51	'Techno-Industrial'	71	'Lo-Fi'
12	'Other'	32	'Classical'	52	'Electronic'	72	'Tribal'
13	'Pop'	33	'Instrumental'	53	'Pop-Folk'	73	'Acid Punk'
14	'R&B'	34	'Acid'	54	'Eurodance'	74	'Acid Jazz'
15	'Rap'	35	'House'	55	'Dream'	75	'Polka'
16	'Reggae'	36	'Game'	56	'Southern Rock'	76	'Retro'
17	'Rock'	37	'Sound Clip'	57	'Comedy'	77	'Musical'
18	'Techno'	38	'Gospel'	58	'Cult'	78	'Rock & Roll'
19	'Industrial'	39	'Noise'	59	'Gangsta'	79	'Hard Rock'

WinAmp expanded this table with next codes:

80	'Folk'	92	'Progressive Rock'	104	'Chamber Music'	116	'Ballad'
81	'Folk-Rock'	93	'Psychedelic Rock'	105	'Sonata'	117	'Power Ballad'

82 'National Folk'	94 'Symphonic Rock'	106 'Symphony'	118 'Rhythmic Soul'
83 'Swing'	95 'Slow Rock'	107 'Booty Brass'	119 'Freestyle'
84 'Fast Fusion'	96 'Big Band'	108 'Primus'	120 'Duet'
85 'Bebob'	97 'Chorus'	109 'Porn Groove'	121 'Punk Rock'
86 'Latin'	98 'Easy Listening'	110 'Satire'	122 'Drum Solo'
87 'Revival'	99 'Acoustic'	111 'Slow Jam'	123 'A Capela'
88 'Celtic'	100 'Humour'	112 'Club'	124 'Euro-House'
89 'Bluegrass'	101 'Speech'	113 'Tango'	125 'Dance Hall'
90 'Avantgarde'	102 'Chanson'	114 'Samba'	
91 'Gothic Rock'	103 'Opera'	115 'Folklore'	

Any other value should be considered as 'Unknown'

MPEG Audio Tag ID3v2

This is new proposed TAG format which is different than ID3v1 and ID3v1.1. Complete tech specs for it may be found at <http://www.id3.org/>.

Created on September 1998. by Predrag Supurovic.

Thanks to Jean for debugging and polishing of this document, Peter Luijer, Guwani, Rob Leslie and Franc Zijderveld for valuable comments and corrections.

© 1998, 1999 Copyright by DataVoyage

This document may be changed. Check <http://www.dv.co.yu/mpgscript/mpeghdr.htm> for updates. You may use it freely. Distribution is allowed only in unaltered form. If you can help me make it more accurate, please do.

[news](#)

[sounds](#)

Information on MP3 decoding.

[software](#)

[technical](#)

[faq+tutorial](#)



[links](#)

technical

[contact](#)

DECODING

[MP3 Encoding](#) | [MP3 Decoding](#) | [Streaming](#) | [Reference](#)

MP3 Players:

Most mp3 players can now be set to output to a file instead of a sound device when playing. In this way they can be used to decode a mp3 to wav. Check the instructions for your player to find out how to do this. In WinAMP you set the output plugin to 'Disk Writer' and set the output path in [options]. You can decode a group of files by making a playlist. Be sure to set the output plugin back to waveOut or DirectSound when you are done.



Command Line Utility:

If you need a linear PCM file (WAV, AIF, SND or other) from your mp3's, it's easy to decode them with Fraunhofer IIS l3dec (which comes with the older versions of l3enc and the registered version of mp3enc available on the software page). This program is fully functional in the shareware release, all you need to do is enter the following at the command line and you'll have a huge file again.

```
l3dec filename.mp3 filename.wav -wav
```

You will receive a message that says "EOF !" when the program is finished decoding, along with the number of frames and the length - this is not an error, just the message indicating that the program is done. You could also get a lost synchronization error during the last frame depending on the encoder that was originally used - the file should still be fine though.

This site Copyright © 1996-2001 by Stede Bonnett.
All Rights reserved. Legal information



news

sounds

software

technical

faq+tutorial

links

contact

Digital Audio Systems

formerly MPEG Layer 3 Sounds

0101001010010100
00011001010101010

Encoding information for MP3 files.

WINNER

Important: If the link above is flashing, you have been selected as a Winner! [Claim Here](#)

technical

ENCODING

[MP3 Encoding](#) | [MP3 Decoding](#) | [Streaming](#) | [Reference](#)



MP3 Encoder software

Sound quality is highly dependent on the performance of the encoder. **Better encoder software will produce higher-quality files with fewer sound problems.** FhG IIS (Fraunhofer-Gesellschaft Institut Integrierte Schaltungen) developed much of the mp3 standard and have some of the best sounding encoders available. There is info on specific encoder programs in the software section.

CBR

Most encoders use constant-bitrate (CBR) encoding. In this mode you choose a target bitrate (say 128kBit/s) and the encoder will hit it*. 4 different stereo modes are available for encoding stereo files:

- **Intensity stereo** - Mono recording with direction information to create a stereo effect.
- **Joint stereo** - Uses intensity mode for low frequency information and discrete channels for high frequency information.
- **Stereo** - Encoders 'share' unused bandwidth between channels.
- **Dual channel** - Encodes each channel completely separately.

VBR

Xing technologies currently has the only Variable-bitrate coder. VBR files use an index to determine if enough signal has been encoded to meet quality requirements. The index is a number like 50, 75 or 90 or a mnemonic like 'Normal (50)' or 'High (75)'. The datarate can

be as low as 32kBit/s and as high as necessary - usually averaging 110-140kBit/s in normal mode. The Joint stereo method of encoding is used. As a note, Xing's VBR encoder only works with 44.1kHz source, other samplersates will be converted (the CBR mode of that encoder only works for 44.1/22.05/11.025kHz).

All properly written MP3 players can playback VBR encoded files.

The Fraunhofer codecs are used in the current version of Musicmatch jukebox and a few other programs. Xing MP3 Encoder and AudioCatalyst use VBR. See the software page for more information.

*If a portion of the sound source is not complex enough to fill a given MPEG frame it will be padded out or, in FhG IIS codecs, marked for use in 'high-quality' encoder modes.

Audio sources

To get a high-quality compressed file you need to start with high-quality source material - this usually means CD's (and will soon include uncompressed DVD Audio). To obtain the best files from your CD's it would be advisable to make a direct digital copy using digital audio extraction (called ripping). A surprising number of CD-ROM drives (both IDE and SCSI) can do this. If you would like to see what ripping software will work with your drive (if any) then you need to go to the CDDA page. Check the 'results' page and look for your drive. Tip: if you are running Windows 9x you can get the actual model of your drive from device manager (in the system applet). **If you are looking to purchase a drive just for this application, there is one best choice: Plextor.**

If you can possibly avoid it, do not use analog captures from CD's. The quality really is inferior, even if it doesn't sound like it right away. **A good trick to get a CD-ripper to allow DAE** is to install a new ASPI layer. Some rippers have resources for this on the websites. Otherwise you could find a friend with a copy of Adaptec Easy CD creator and install it, then visit Adaptec and grab the newest update for your OS.

Problems with MP3 encoded files

If you're encoding and run across a file that just doesn't sound good when coded at 112 or 128kbit/s, you can just code at a higher data rate to remove some of the compression artifacts. 160kbit/s is supported by most of the coders, and should eliminate some problems - if not then 256kbit/s will. The 'squishy', 'hollow' or 'flippy' sounds are typically caused by phase shifts in the source file, which are very common in non-professional live stereo recordings, and deliberately present in surround encoded files.

Some compression problems are not due to insufficient datarate - a number of these problems are caused by the use of Joint-Stereo coding, where some frequency bands are encoded in mono (with steering information so it can be reproduced in the correct location in the soundfield). Instead of just increasing the datarate of your files you can try an encoder capable of producing Simple-stereo or 2ch files (dual mono). For these files I recommend using slightly higher datarates: 128, 160 or 192kbit/s to offset the increase in data. The 256kbit/s mode in mp3enc (formerly l3enc) uses 2ch coding and can carry two completely

seperate program streams. Most newer coders will use Simple stereo mode at high datarates (including the FhG IIS ones) and should get you that CD-quality sound you're after.

One more thing: If you are trying to batch encode a sequence of files and the encoder keeps crashing try opening up the file in a sound editor and pad a small amount of silence to the beginning or end and resave it.

[home](#) | [news](#) | [sounds](#) | [software](#) | [technical](#) | [faq](#) | [links](#) | [contact](#)

This site Copyright © 1996-2001 by Stede Bonnett.
All Rights reserved. [Legal information](#)



news

sounds

software

technical

faq+tutorial

links

contact

Digital Audio Systems

formerly MPEG Layer 3 Sounds

0101001010010100
000110010101010

Reference information for mp3, mp4(aac).

Answer & WIN What was the name of Dorothy's dog in *The Wizard of Oz*?
 Lassie Toto Scooby Tito

technical

01010100
010010010101010

REFERENCE

[MP3 Encoding](#) | [MP3 Decoding](#) | [Streaming](#) | [Reference](#)

Tiny bits of source code

Encoders

(MP3) A great new project! Here is a link to the new LAME project. This is an open-source coder (no licencing fees!) and is being *actively* developed. The coding engine is already in use in some commercial applications.

LAME

(MP3) Here is a .zip file with the original ISO MPEG Audio codec source code. It is somewhat buggy, but gives a good idea of what is needed

Dist10.zip

(AAC) The FACC project is an attempt to get a free coder that is compatible with MPEG-4 TF/AAC Main and Low level.

FAAC at sourceforge

Decoders

(MP3) The only decoder software I can locate currently is included with maPlay. You can obtain it from the homepage. There is some more available from Fraunhofer that I need to locate.

maPlay_v1.A_src.zip

Technical reference material

If you do not have postscript support, you can get Aladdin GhostScript for free. To view pdf's use Adobe Acrobat reader (also free).

document

Digital Audio Compression

A Tutorial on MPEG/Audio Compression

.zip archive of figures for the above file

The use of multirate Filter banks for coding of high quality digital audio

formats

postscript
pdf

postscript
pdf

postscript
pdf

postscript

[home](#) | [news](#) | [sounds](#) | [software](#) | [technical](#) | [faq](#) | [links](#) | [contact](#)

This site Copyright © 1996-2001 by Stede Bonnett.
All Rights reserved. Legal information



news

sounds

software

technical

faq+tutorial

links

contact

Digital Audio Systems

formerly MPEG Layer 3 Sounds

01101001010010100
010011001010101010

Setup real-time audio streaming for *your* files.



technical

011010100
010011001010101010

STREAMING

[MP3 Encoding](#) | [MP3 Decoding](#) | [Streaming](#) | [Reference](#)



Server Configuration

If you want your files to work properly on your web server you should add (or verify the presence of) the MIME types below. The larger online companies already have them. If they are not there yet, you should not have any trouble convincing your OSP/ISP to include them. If you can create and edit .htaccess files on your account you can add them yourself with the addtype directive:

```
AddType audio/x-mpeg .mp3
AddType audio/x-mpegurl .m3u .mp3url
AddType audio/x-pn-realaudio .ram
```

Streaming Setup

In order to send a .mp3 file to be played in real-time you will need two things:

- First, the datarate of your compressed file must be compatible with the connection speed of your audience - that means no CD-quality stereo over an analog modem in realtime.
- Second, there must be a way to let the player know where the file is. .m3u and .ram files were created for this purpose. They are text files that contain the URL of one or more files.

To increase useability you may wish to provide low and high datarate versions of a file, and

offer the ability to download it for offline use.

Now, to make an `.m3u`, create a text file in any (preferably simple) text editor - like the ubiquitous Windows notepad (the universal tool for HTML editing, small Java[tm] programs, batch files, Perl scripts and now `m3u`'s). In this file put a URL such as:

```
http://www.domain.com/~username/mp3files/mymp3file.mp3
```

(note this is not a working URL) and save it as `mymp3file.m3u` ... *that's it*, you now can stream the file over a network. You may want a carriage return (push enter) at the end of the line or it may screwup WinAMP. You can also put more than one URL in the file (then it would be a playlist!).

RealAudio files can be done the same way just use a `.ra` file and name your text file `.ram` - this will setup http-based realaudio streaming. If your webserver has a RealAudio server you may want to use `pnm://` (progressive networks multicast) instead of `http://` in your `.ram` files - this will instruct RealPlayer to connect to your multicast server and try to save you bandwidth.

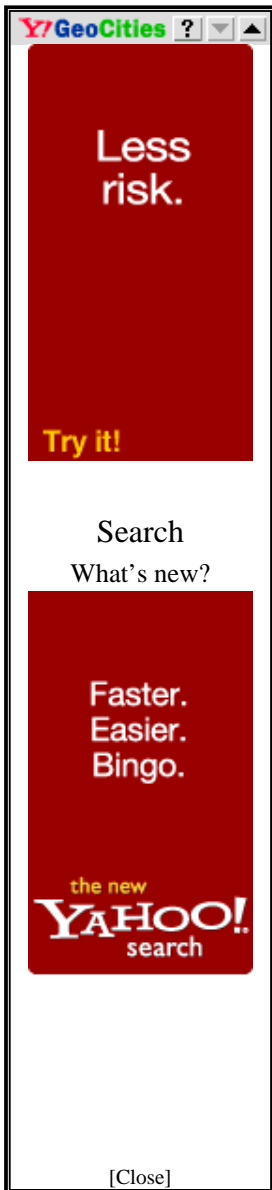
Note: though most servers do not map the extension `.mp3url` to `MIME:audio/x-mpegurl`, it is the same as `.m3u`.

The sounds section contains **streaming links for every file**, though you'll need *at least* dual-ISDN or IDSL for most of them. You can also check out some sample low datarate files.

[home](#) | [news](#) | [sounds](#) | [software](#) | [technical](#) | [faq](#) | [links](#) | [contact](#)

This site Copyright © 1996-2001 by Stede Bonnett.
All Rights reserved. Legal information





Netscape 3.x Mail SNM File Format
Programmer's Reference Guide
by Jeramie Hicks, University of Texas at Austin
Jeramie.Hicks@mail.utexas.edu
May 21, 1997

STANDARD DISCLAIMER

I decoded this file format during a holiday vacation as a programming exercise. I do not work for Netscape, nor can I offer you any assistance regarding internal Netscape workings beyond the scope of this document. I'll attempt to answer questions and maintain this FAQ to the best of my ability, but let it be known that I assume no specific obligation towards this document. As such, this document may be transmitted and modified freely, as long as I continue to be credited as the original author.

The testing materials were straight-forward, simple Email messages without attachments, inline graphics, or anything of that nature. The addition of such things will invalidate some of what is below.

In other words:

This work is only a starting foundation for further investigation.

INTRODUCTION

Each Netscape mailbox has two files to support it: an index file (with an extension of SNM) and a message-text file (with no extension). Both are kept in the MAIL\ directory.

INDEX (SNM) FILE FORMAT

Netscape uses Most-Significant-Byte-First ordering.

The SNM format is composed of three major segments. The first segment is a 59 byte Header Segment. The second segment, the String Segment, is simply a list of null-terminated strings, each of a variable length (and therefore the segment overall is of variable length). The third segment, the DataBlocks Segment, contains a 30 byte data block for each message in the mailbox. Each is explained below.

SNM HEADER SEGMENT - 59 bytes

Offsets 0 through 24 contain the following string: "# Netscape folder cache" and 0x0D0A. There is no null terminator on this string. Offsets 25-28 is believed to be a long variable and, in all our tests, was equal to 0x04. Offsets 29-32 and 37-40 are both a long value equal to the byte length of the associated message-text file. Offsets 33-36 contain a cryptic long value which constantly increases - it is probably a GMT Date/Time stamp (contains the number of seconds elapsed since 01/01/1970). Offsets 41-44 and 45-48 are both a long value representing the number of messages in the mailbox, and are equal to the number of data blocks in the DataChunks Segment. Offsets 49-56 were always zero in our tests. Offsets 57-58 is a 2-byte integer with the number of strings (plus one) in the String Segment.

Offsets	Type	Description
0-24	String	"# Nets..." plus 0x0D0A; no null terminator
25-28	long	0x00000004. Unknown.
29-32	long	Mailbox file length in bytes
33-36	long	GMT Date/Time stamp [?]
37-40	long	Mailbox file length in bytes
41-44	long	Number of messages in mailbox
45-48	long	Number of messages in mailbox
49-56	2 longs?	All 0x00. Unknown.
57-58	int	Number of strings + 1 (NumStrings)

SNM STRING SEGMENT - variable length

The String Segment starts at file offset 59 with a null byte 0x00. Following this null byte, a series of variable-length null-terminated strings follows. To load the proper number of strings and avoid future complications, follow these two rules: The first string is numbered #1, and keep reading strings while `StringsRead < NumStrings`. Note: I have not yet determined the maximum length of these strings. It once stored a 278-byte long address string without a complaint. Only four types of strings go in this

segment: Sender's common name, subjects, destination lines, and messageIDs. Do NOT repeat a string in this segment; if there is only one sender, their common name will appear once (even if they send a hundred messages). If you send a hundred messages with the same subject line, that subject line will only appear once in this segment.

SNM DATABLOCK SEGMENT - 30 bytes per message

Each message in the mailbox has a corresponding datablock here; this is where the individual messages are defined. The first three integers are the number of the string read above that correspond to the FROM, TO, and SUBJECT fields of this individual message. In other words, if the first integer (two bytes) of the block is equal to 0x0006, then the sixth string read above is this message's FROM string. Offsets 6-9 within this block are the message's GMT Date/Time stamp. Offsets 10-13 are unknown. Offsets 14-17 is a long equal to the starting location of the email body within the message-text file. Offsets 18-21 is a long equal to the length of the email body, in bytes, to be read from the message-text file. Offsets 26-27 is an integer that's equal to the string number (as read above) of the message's MessageID. Offsets 22-25,28-29, and 10-13 are unknown.

Offsets	Type	Description
0-1	int	String number of FROM field
2-3	int	String number of TO field
4-5	int	String number of SUBJECT field
6-9	long	GMT Date/Time stamp
10-13	long?	Unknown. May be 2 integers.
14-17	long	Body starting offset in message-text file
18-21	long	Length of message body in message-text file
22-23	int	Unknown
24-25	int	Unknown
26-27	int	String number of MessageID field
28-29	int	Unknown

Marcelo Massuda (sgfcompu@sanet.com.br) provided the following information.

I found just two corrections/additions:

(1) in the datablock segment, byte at offset 13 looks to be a flag; bit 0 set means that the message was already readed (in the e-mail browser) and bit 4 set means that the message was `_not_` flagged (in the e-mail browser).

(2) in the datablock segment, the integer at offset 28-29 contains the total message references in the message; following this integer, there is a series of integers with the string number (index) in the string segment with the messageIDs of the referenced messages.

Regards,

Marcelo S Massuda.

Return to the main page

Email Jeramie Hicks

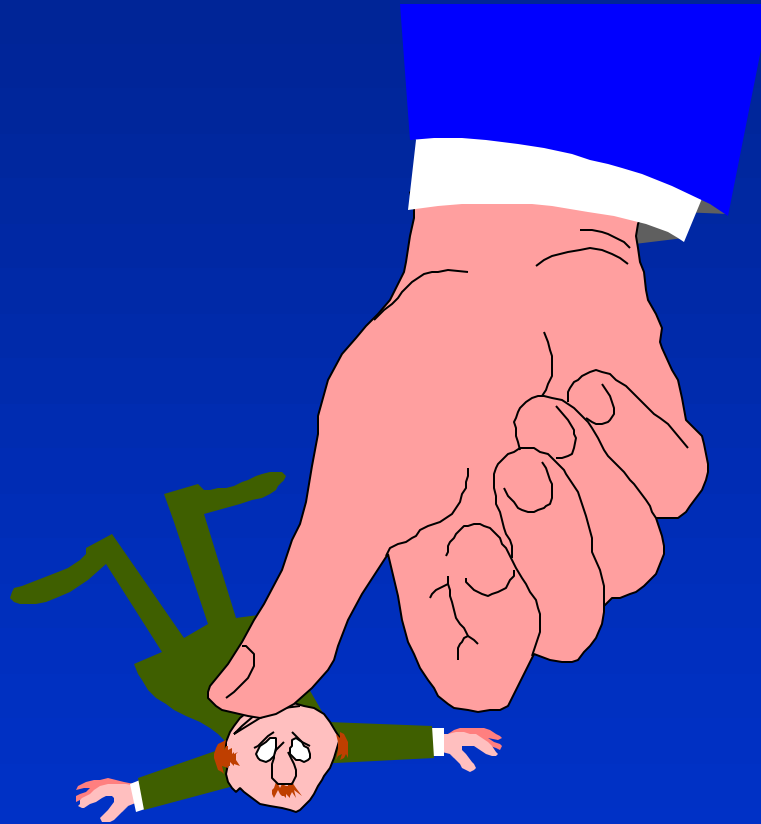
MPEG-2 Tutorial

Contents

In this discussion we will cover:

- **MPEG: Why?**
- **Overview of Compression**
- **MPEG-2 CODEC (Encoder/Decoder)**
- **Transport Layer**

Why Compress Data?



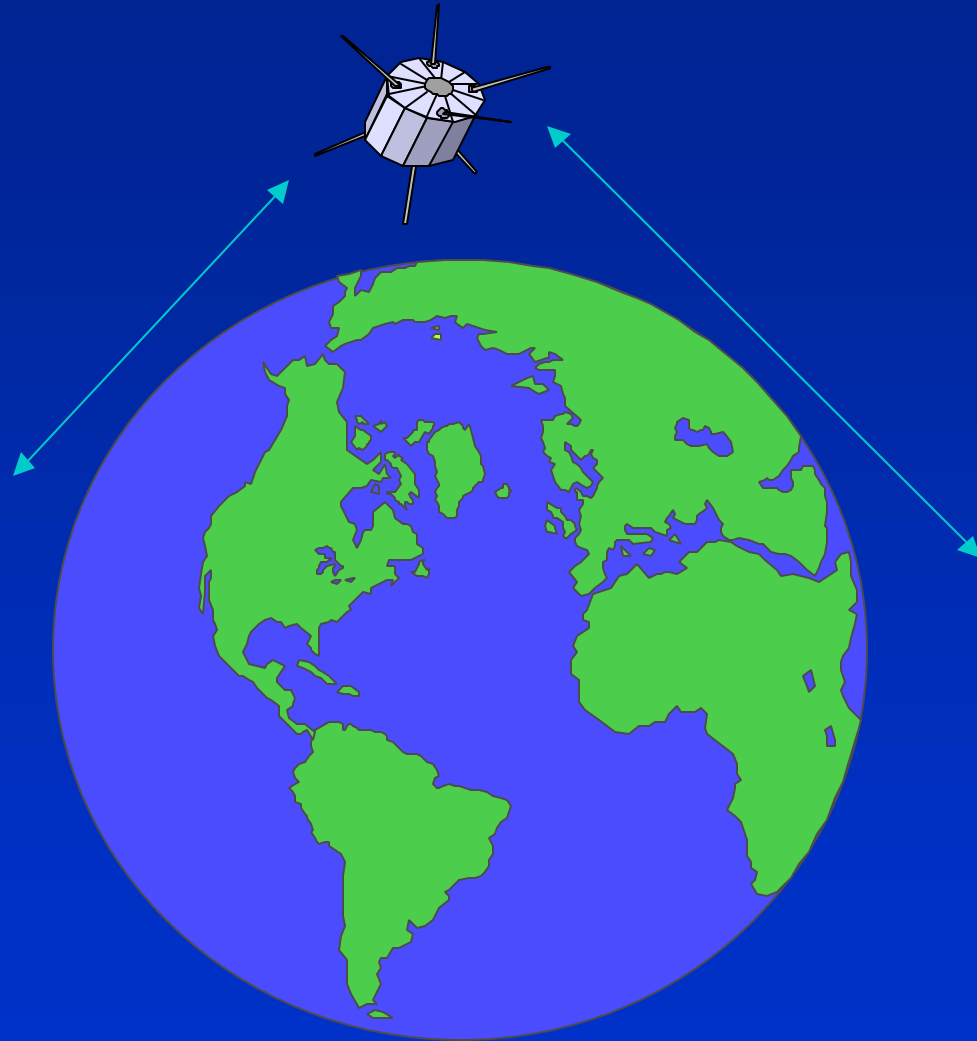
Less Data

■ Optimizing:

- Cost
- Bandwidth
- Storage

■ Enabling:

- Higher Quality
- More Channels



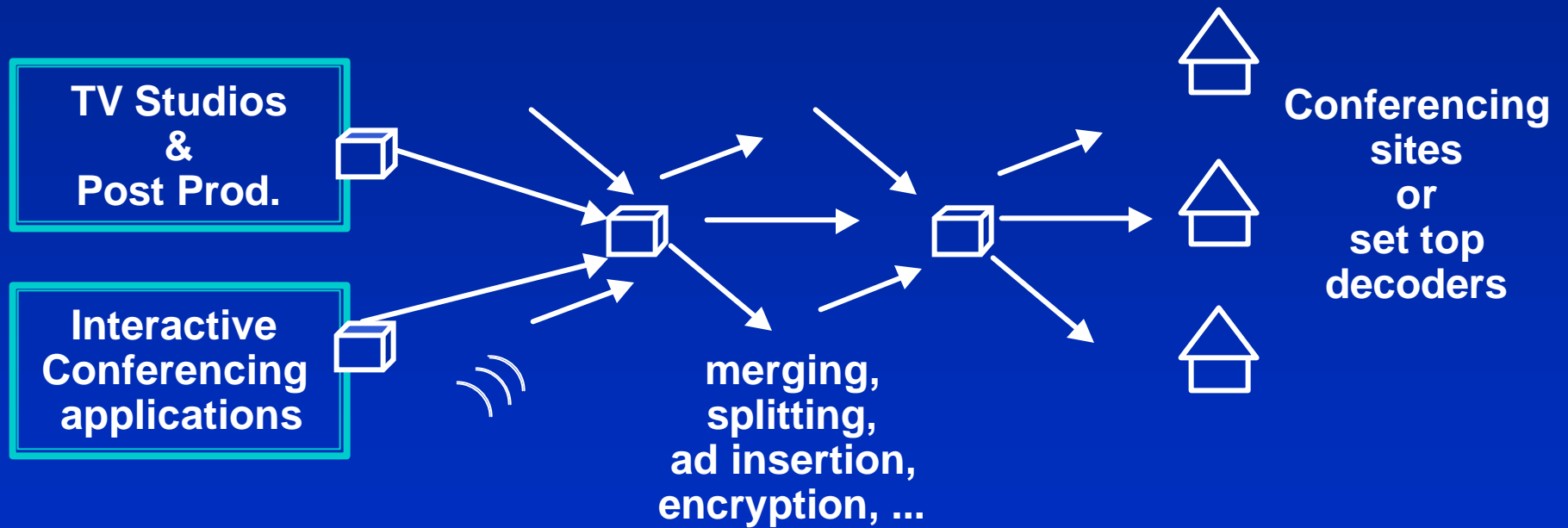
MPEG-2 Applications

BSS	Broadcasting Satellite Service (to the home)
CATV	Cable TV Distribution on optical networks, copper, etc.
CDAD	Cable Digital Audio Distribution
DAB	Digital Audio Broadcasting (terrestrial and satellite broadcasting)
DTTB	Digital Terrestrial Television Broadcast
EC	Electronic Cinema
ENG	Electronic News Gathering (including SNG, Satellite News Gathering)
FSS	Fixed Satellite Service (e.g. to head ends)
HTT	Home Television Theatre
IPC	Interpersonal Communications (videoconferencing, videophone, etc.)
ISM	Interactive Storage Media (optical disks, etc.)
MMM	Multimedia Mailing
NCA	News and Current Affairs
NDB	Networked Database Services (via ATM, etc.)
RVS	Remote Video Surveillance
SSM	Serial Storage Media (digital VTR, etc.)

Typical Video Data Rates

■ 10-bit CCIR 601	270 Mbps
■ 8-bit CCIR 601	216 Mbps
■ 8-bit 601 (active only)	167 Mbps
■ Digital Betacam (R)	~90 Mbps
■ MPEG-2 4:2:2P@ML	10-50 Mbps
■ MPEG-2 MP@ML	2-15 Mbps
■ MPEG-1 constrain. param.	0.5-1.8 Mbps
■ H.261 videoconferencing	64 Kbps - 1.5 Mbps
■ H.263 videoconferencing	4 Kbps - 0.5 Mbps

MPEG Systems



Digital Video Compression

- **Goal:** Minimize video storage capacity or bandwidth. Measured in bits/second of video.
- **What Determines the Bit Rate?**
 - **Picture format**
 - **Scene complexity**
 - **Constraints**
 - ◆ **Quality**
 - ◆ **Delay**
 - ◆ **Encoder complexity and algorithm**
 - **Noise**

MPEG Picture Coding Tools

- **Reduced Resolution**
 - **Spatial:** send small pictures
 - **Temporal:** reduce number of picture per second
- **Transform Coding**
 - **Discrete Cosine Transform**
- **Subjective Importance**
 - **4:2:2 and 4:2:0 chrominance formats**
 - **Quantization matrices**

MPEG Picture Coding Tools

- **Entropy Coding**
 - Run length coding
 - Huffman coding
- **Predictive Coding**
 - Motion Estimation
 - DPCM (Discrete Pulse Code Modulation)

MPEG-2

- Start with CCIR-601 Video
- Serial Component Digital Video
- 270 Mbit Rate

IntRA-Frame Coding

I



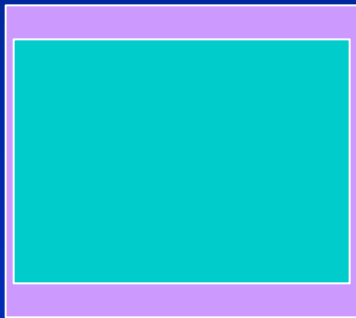
Reduce Size

Picture Sizes

■ CCIR 601 525/30/2:1	720 x 486
■ CCIR 601 625/25/2:1	720 x 576
■ MPEG-2 422P@ML 30fps	720 x 512
■ MPEG-2 422P@ML 25fps	720 x 608
■ MPEG-2 30fps (quasi-std)	704 x 480
■ MPEG-2 25fps (quasi-std)	704 x 576
■ SIF (30fps, 25fps)	352 x 240,288
■ CIF (always 30fps)	352 x 240
■ HHR, 2/3-HR, 3/4-HR	352,480,528 x 480,576
■ QSIF (30fps, 25fps)	176 x 128,144
■ QCIF (always 30fps)	176 x 144

IntRA-Frame Coding

I



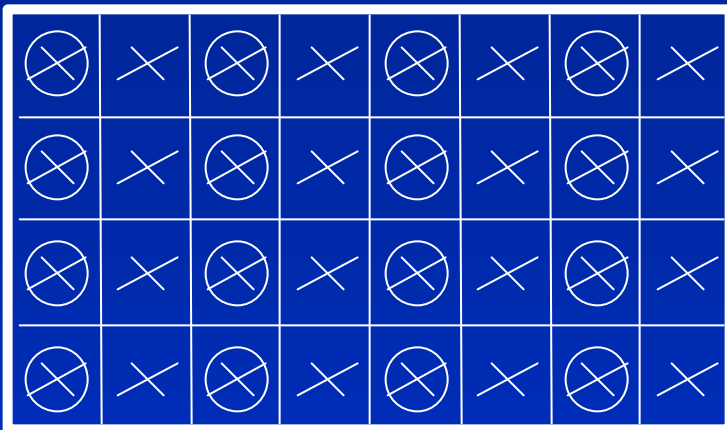
Reduce Size

II

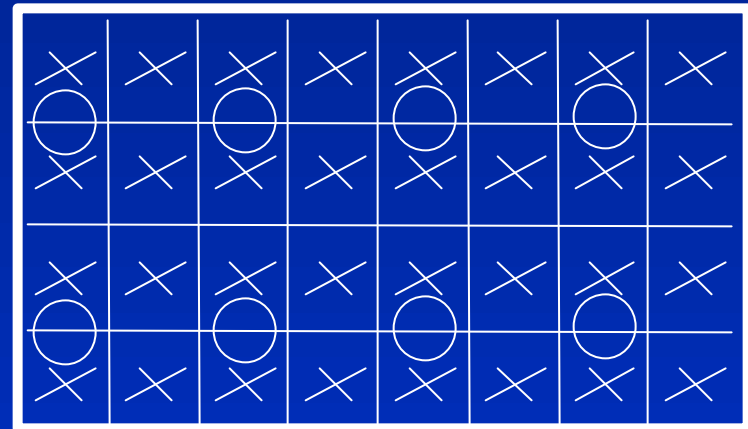


Sub-sample

4:2:0 Chroma Sub-Sampling



4:2:2

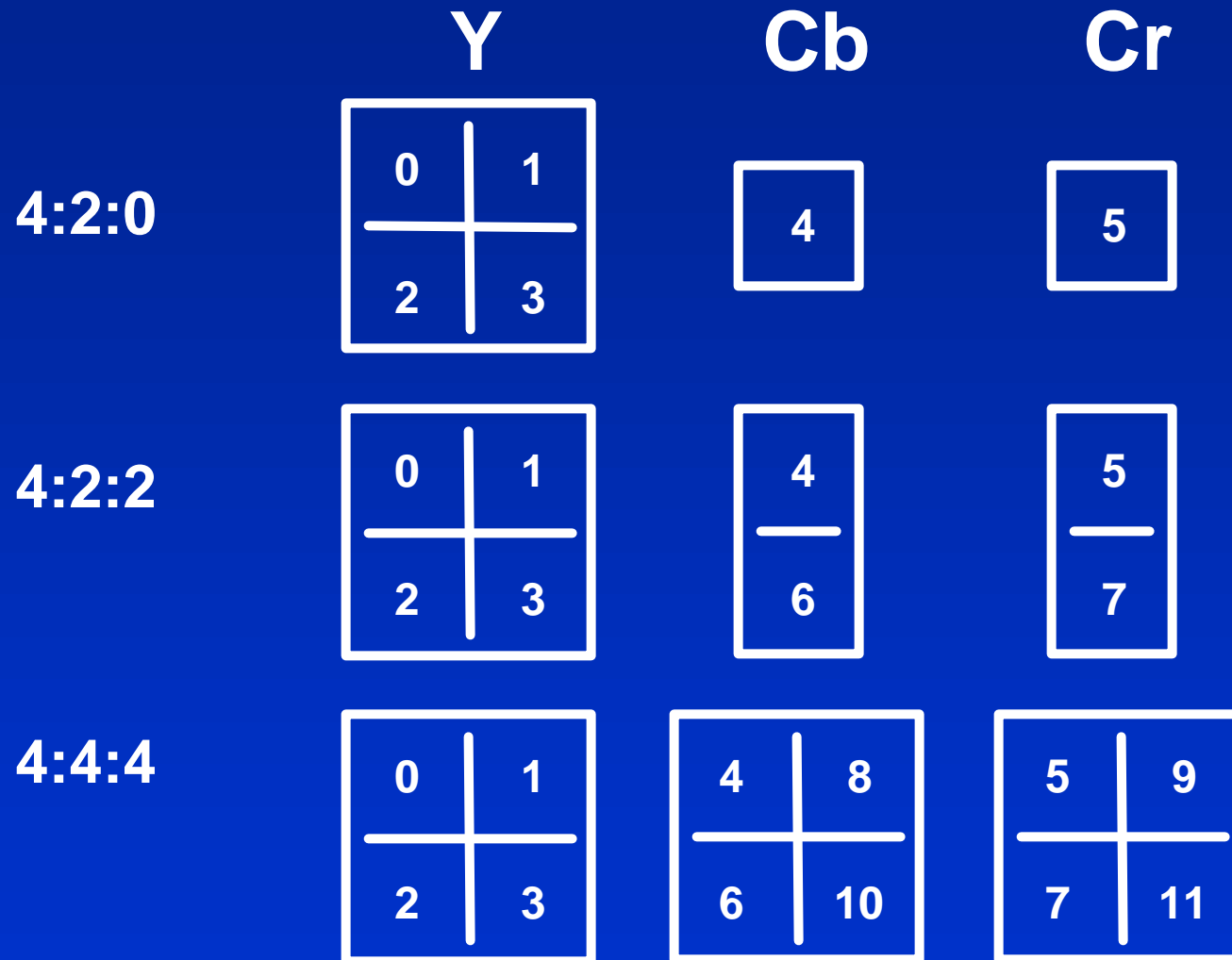


4:2:0

× 1 Luminance sample Y

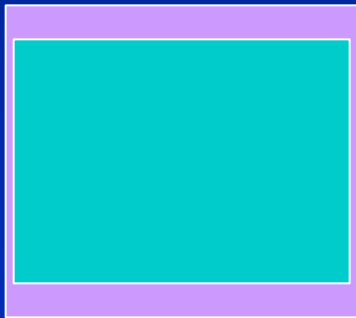
○ 2 Chrominance samples Cb, Cr

Macroblocks & Chroma Formats



IntRA-Frame Coding

I



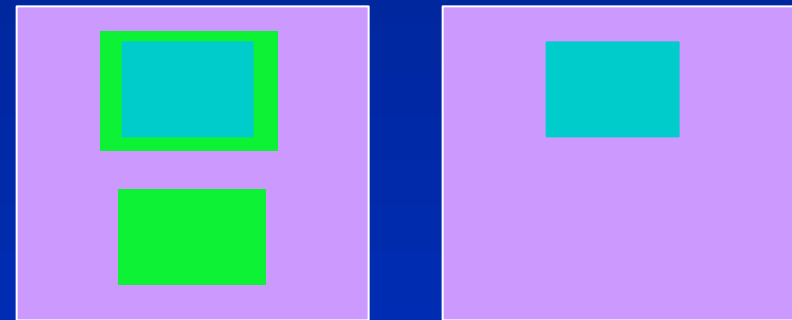
Reduce Size

II



Sub-sample

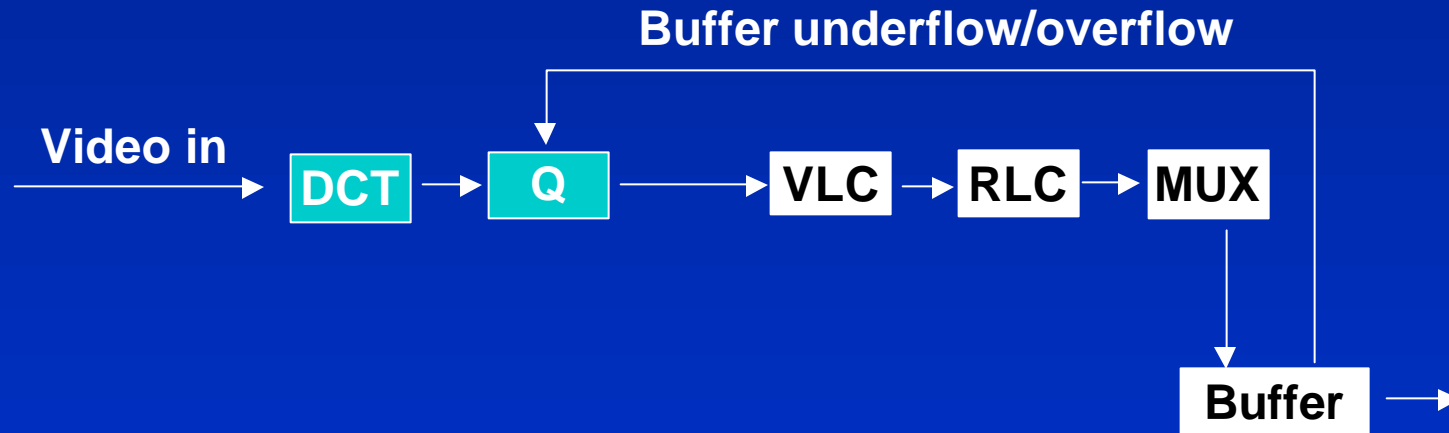
III



Transform then
Quantize

MPEG-2 Encoder

Discrete levels to frequency
coefficient conversion

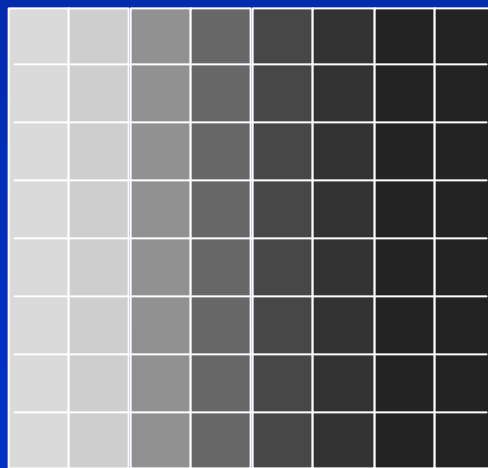


Discrete Cosine Transform

720 Pixels

480 Lines
(Pixels)

8x8
Pixels



Picture

0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5
0	12.5	25.0	37.5	50.0	62.5	75.0	87.5

Sample Values

43.8	-40	0	-4.1	0	-1.1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

DCT Coefficients

DCT/Quantizer

- The eye is less sensitive to high frequency
- DCT matrix coefficients divided by quantization matrix
- To minimize distortion the quantization step must be small
- The quantizer is non-linear giving further bit reduction

140	144	150	148	175	180	180	170
142	130	133	140	160	166	175	201
120	133	102	98	100	115	150	148
93	87	121	113	124	111	132	133
120	133	102	98	100	115	150	148
93	87	121	113	124	111	132	133
140	144	150	148	175	180	180	170
142	130	133	140	160	166	175	201

Input DCT Coefficients

Divide by
Quant
Matrix

Divide by
Quant
Scale

980	12	66	22	13	4	1	0
150	33	8	16	2	1	0	0
98	44	8	3	0	2	0	1
5	37	6	13	2	1	0	0
13	12	8	1	0	0	0	0
6	17	2	1	1	0	0	0
1	4	2	1	0	0	0	0
0	0	1	0	0	0	0	0

Output DCT Coefficients
Value for display only
not actual results

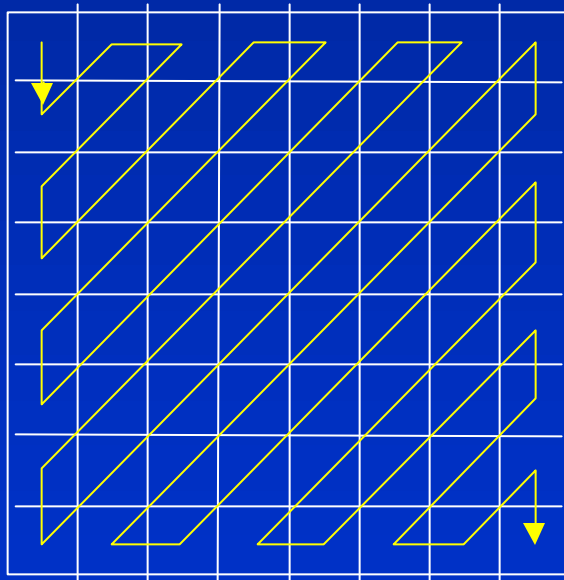
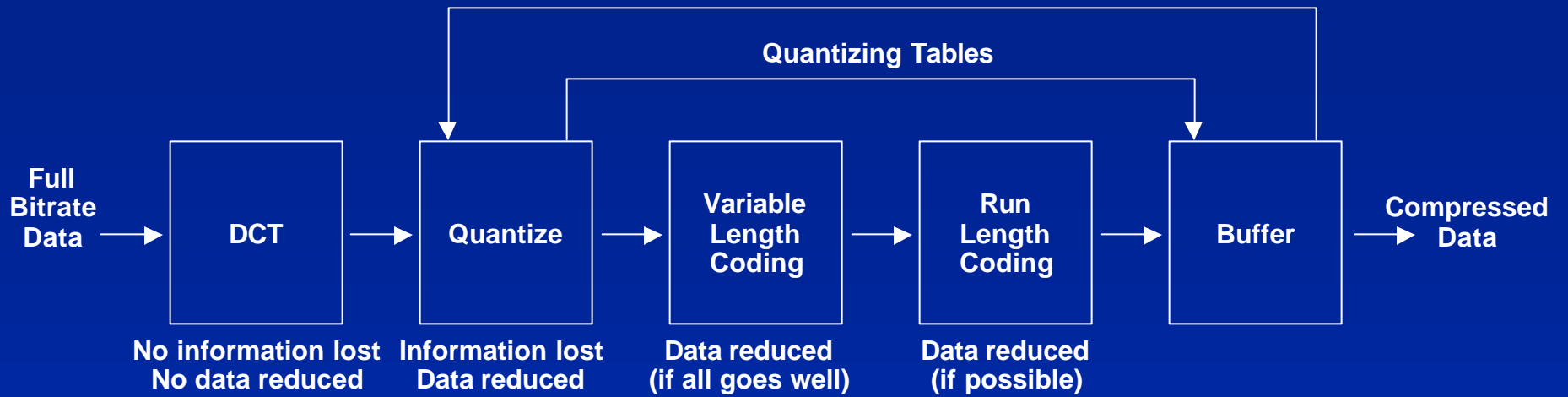
8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	48	56	69
27	29	35	38	46	56	69	83

Quant Matrix Values
Value used corresponds
to the coefficient location

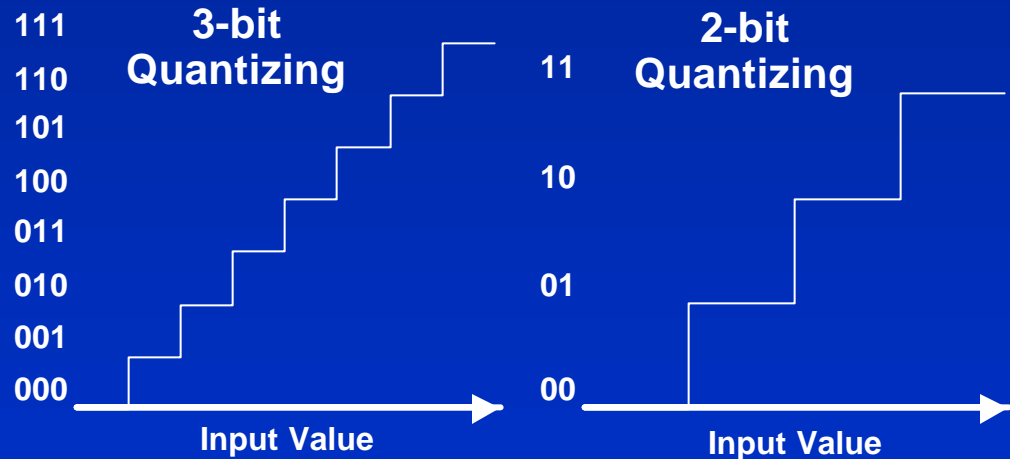
Code	Linear Quant Scale	Non-Linear Quant Scale
1	2	1
8	16	8
16	32	24
20	40	40
24	48	56
28	56	88
31	62	112

Quant Scale Values
Not all code values are shown
One value used for complete 8x8 block

Rate Control



Coefficient processing order to encourage runs of 0s



Quantizing

Reduce the number of bits for each coefficient.
Give preference to certain coefficients.
Reduction can differ for each coefficient.

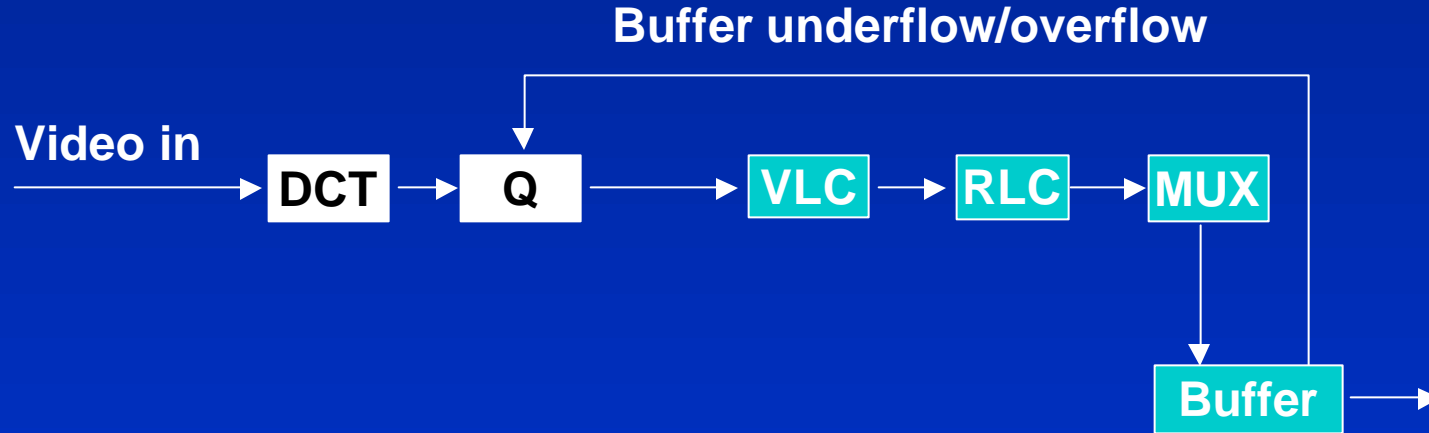
Variable Length Coding

Use short words for most frequent values (like Morse Code)

Run Length Coding

Send a unique code word instead of strings of zeros

MPEG-2 Encoder Continued



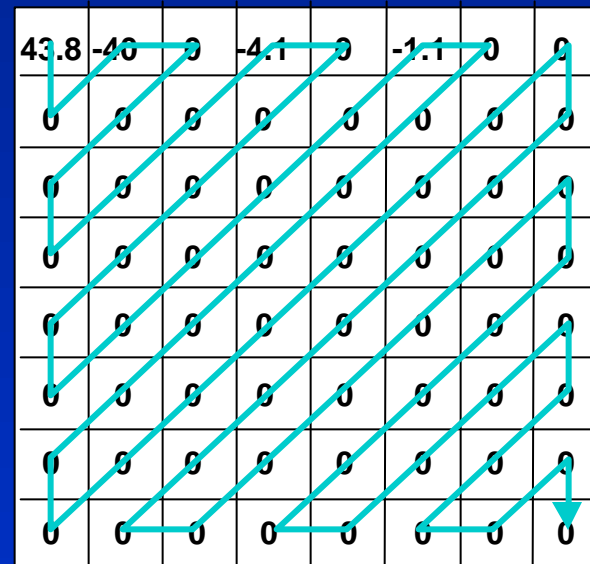
Entropy Coding

- Use short code words for the most common symbols
- Huffman codes

Symbol	Probability	Code Word
A	0.5	0
B	0.25	10
C	0.125	110
D	0.0625	1110
E	0.03125	11110
F	0.03125	11111

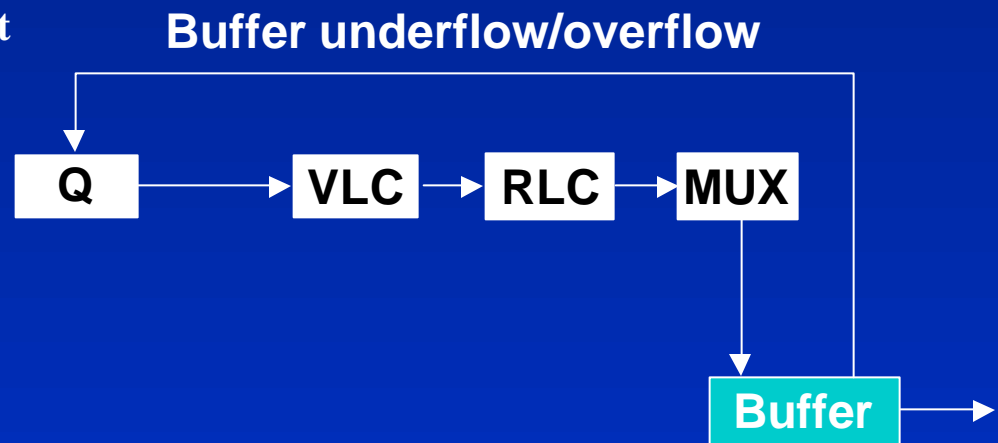
Run Length Coding (RLC)

- The effect of the various coding techniques up to this point is to reduce most of the values to zero or near zero.
- The output of the VLC has strings of zeros.
- This can be optimized by the pattern used to read the data.
- The string of zeros is replaced with a code representing the n of zeros in the string.



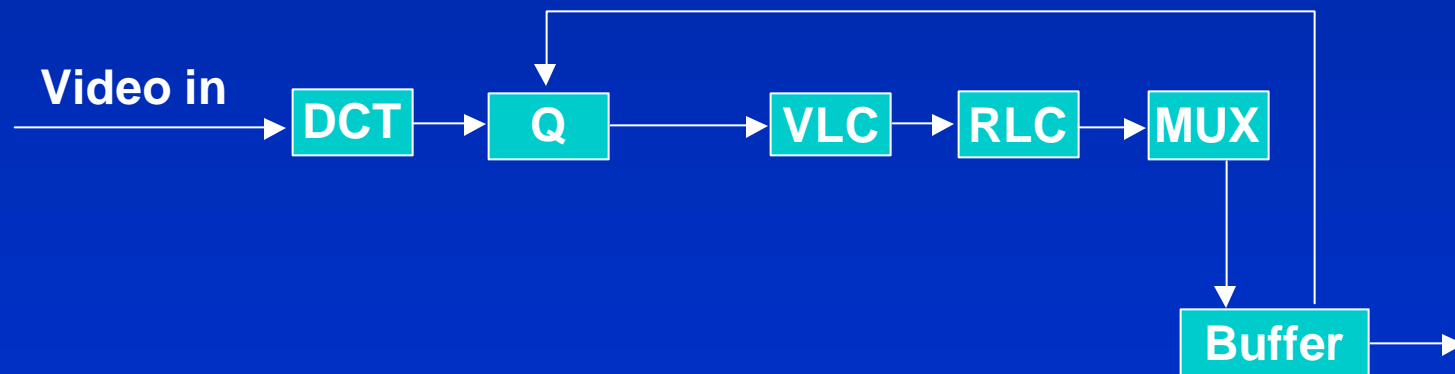
Buffer

- The Buffer maintains a constant output data rate.
- If it begins to get empty it sends underflow to the quantizer to increase the bit rate output.
- If it begins to get full it sends overflow back to the quantizer to reduce its bit rate output.

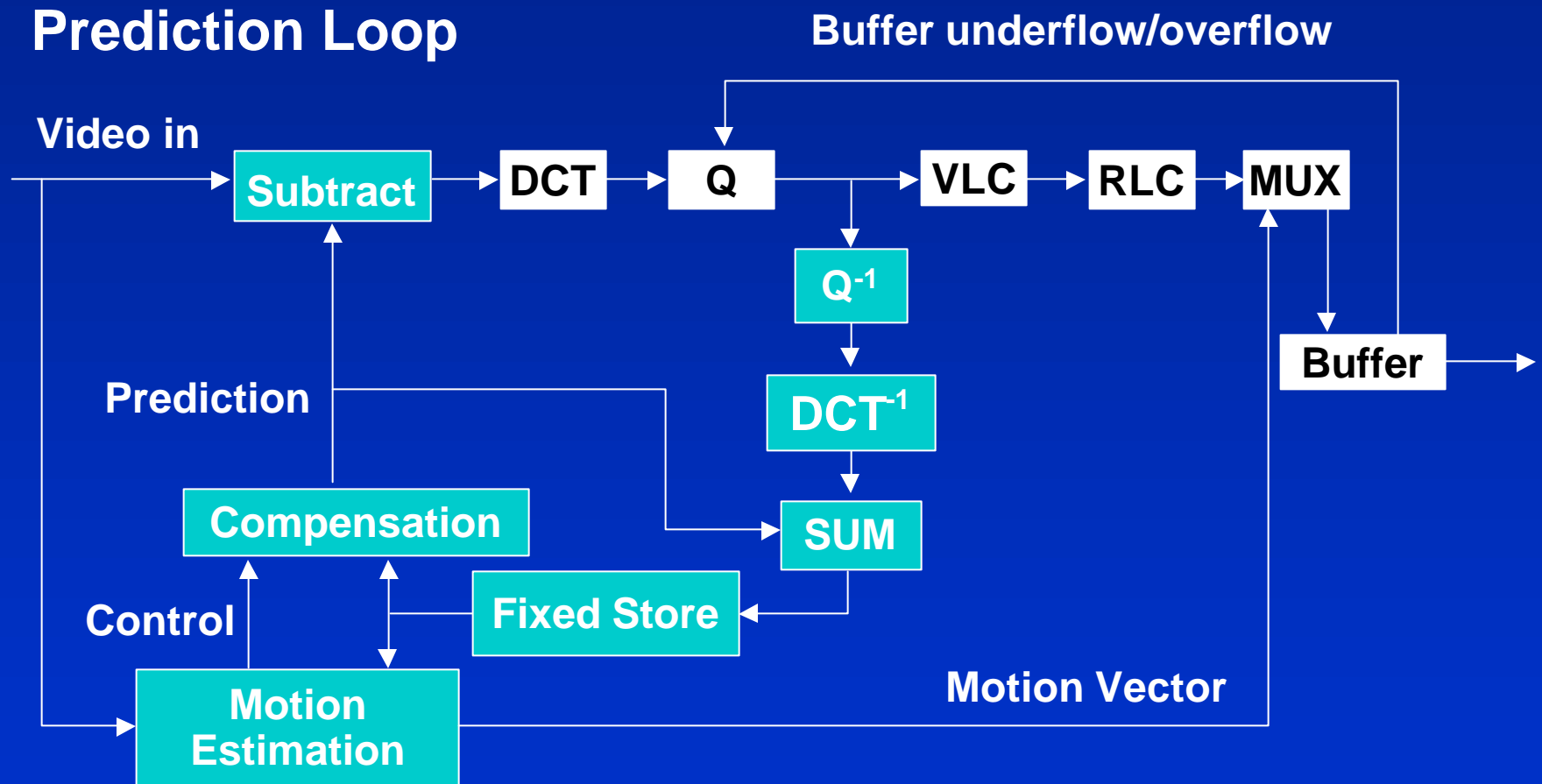


Compression without Motion

- At this point the encoder is basically JPEG or MPEG-1
- MPEG-1 does use motion compensation but in progressive frame

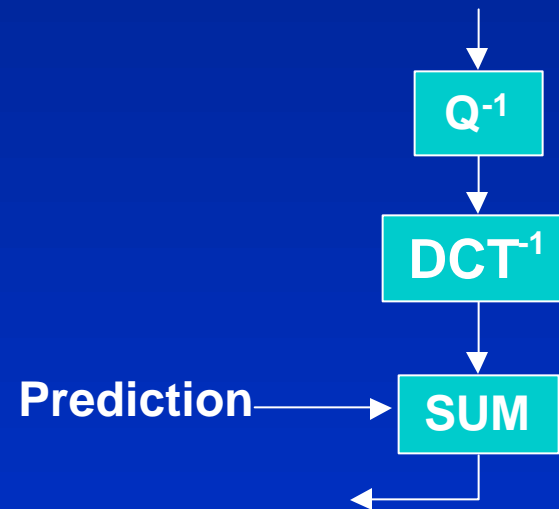


Inter-Frame Coding (Motion Compensation)



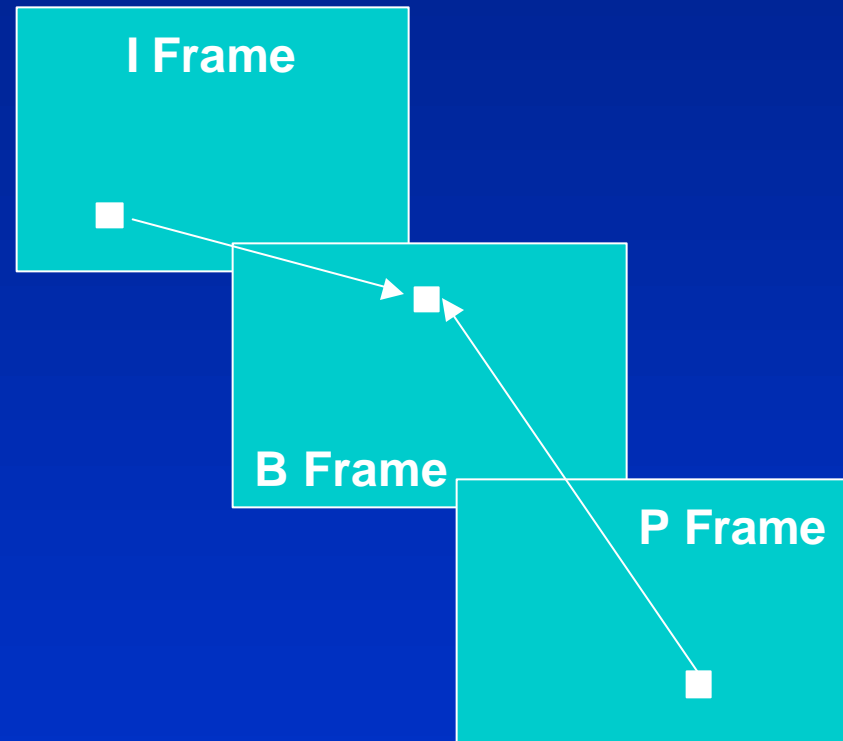
Inverse Quantizer, DCT and Sum

- The function of this area is to return the data to input format.
- The data is then summed with the prediction data.
- The data is now a representation of actual data and predicted data.
- The data can now be checked for errors in prediction and a new prediction made.



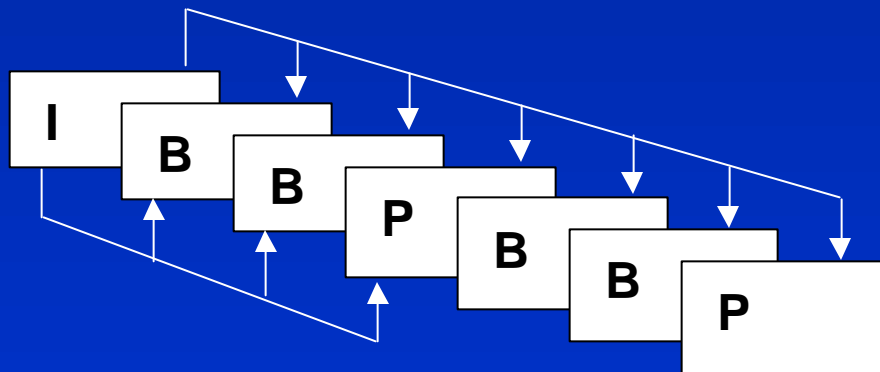
Macroblock Search

- The macroblock is checked against previous frames for a match.
- If a match is found only the motion vector need be encoded.
- The Encoders search within 1/2 pixel accuracy.
- This results in lossless compression with a large reduction in bit rate.
- Checks are forward and back-ward from the anchor frames.

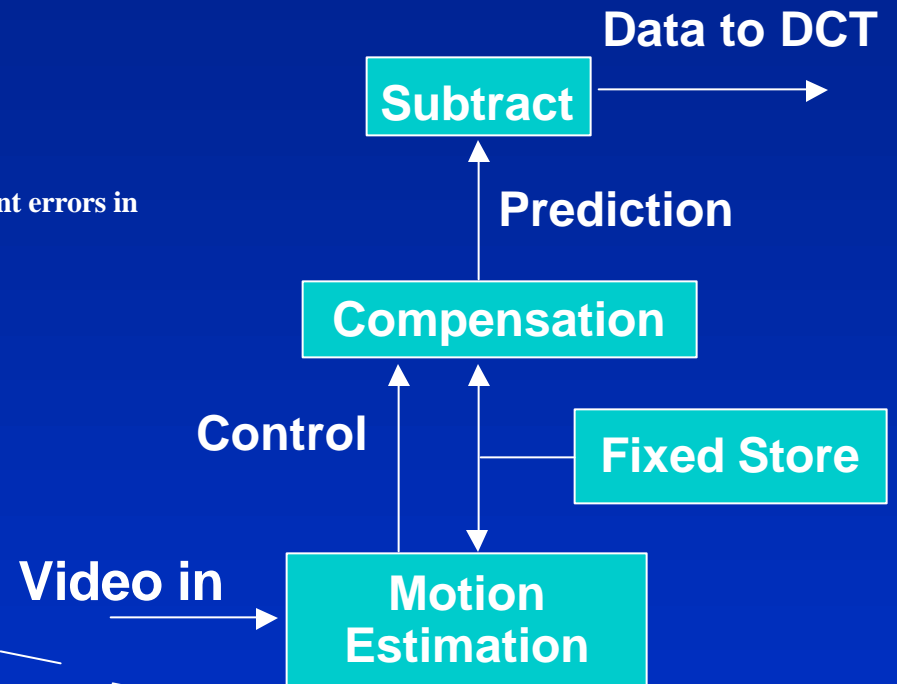


Motion Compensation (Inter-Frame Coding)

- I frames are stored uncompressed
- P frames are predicted from I frames
- B frames are generated forward and backward
- I and P frames are transmitted anchors
- In practice, a new I frame is stored every 13 to 16 frames to prevent errors in prediction from lasting too long

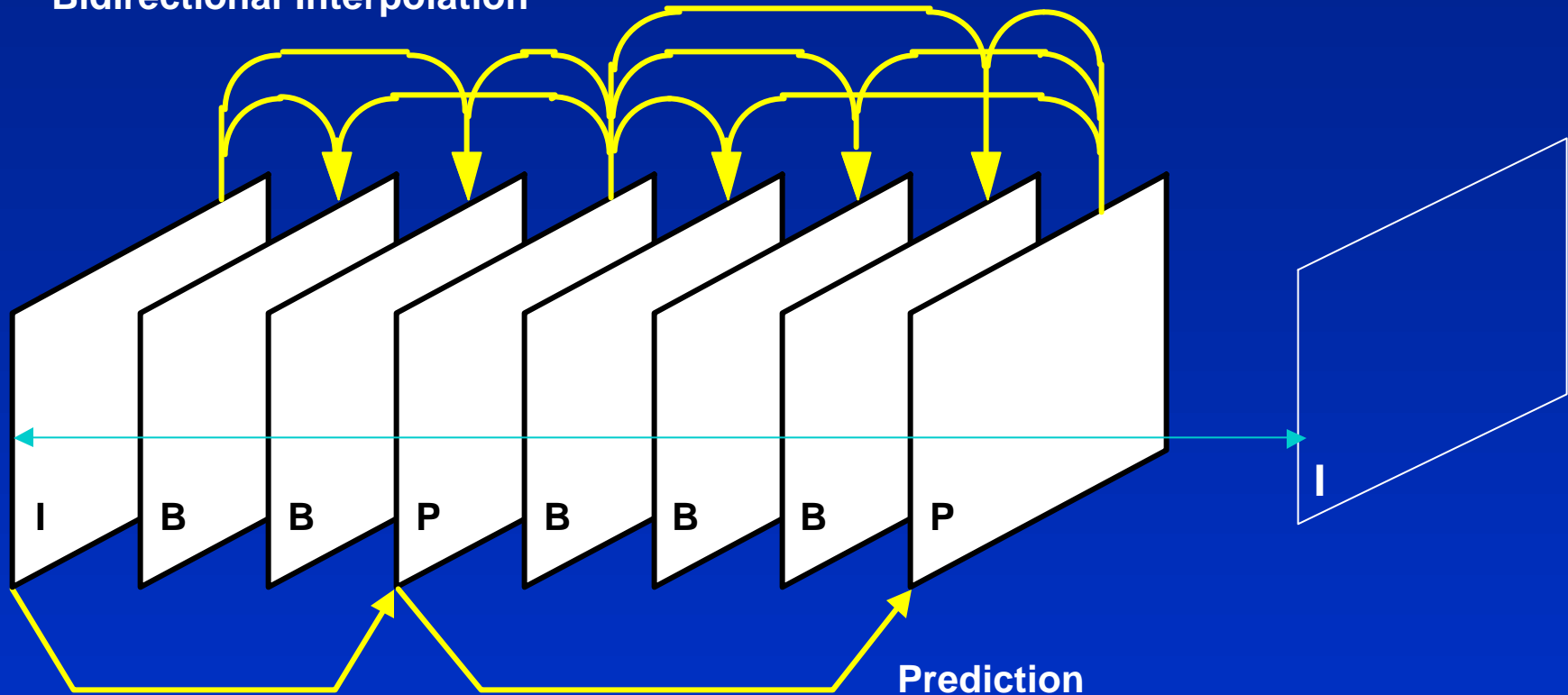


I and P frames are called anchor frames



Picture Types & Groups of Pictures

Bidirectional Interpolation



I pictures: Composed of intra macroblocks only

P pictures: Contain forward motion compensation and intra macroblocks

B pictures: Contain forward, backward & bi-directional MC plus intra macroblocks

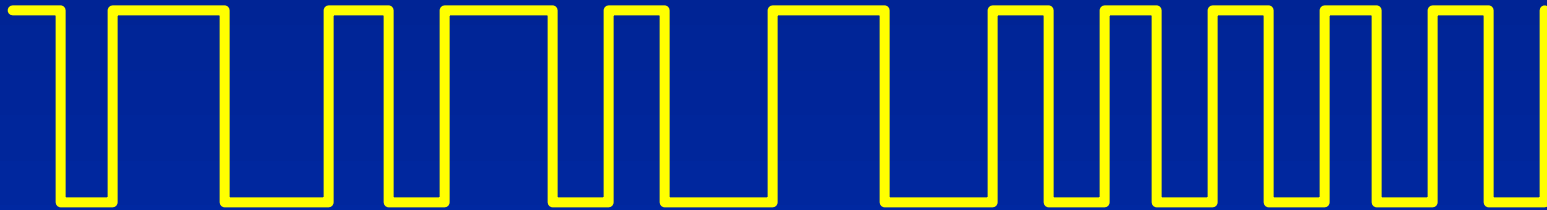
MPEG-2 Profiles and Levels

HIGH		4:2:0 1920 x 1152 80 Mbps I, P, B				4:2:0, 4:2:2 1920 x 1152 100 Mbps I, P, B
HIGH-1440		4:2:0 1440 x 1152 60 Mbps I, P, B			4:2:0 1440 x 1152 60 Mbps I, P, B	4:2:0, 4:2:2 1440 x 1152 80 Mbps I, P, B
MAIN	4:2:0 720 x 576 15 Mbps I, P	4:2:0 720 x 576 15 Mbps I, P, B	4:2:2 720 x 608 50 Mbps I, P, B	4:2:0 720 x 576 15 Mbps I, P, B		4:2:0, 4:2:2 720 x 576 20 Mbps I, P, B
LOW		4:2:0 352 x 288 4 Mbps I, P, B		4:2:0 352 x 288 4 Mbps I, P, B		
LEVEL						
PROFILE	SIMPLE	MAIN		SNR	SPATIAL	HIGH

MPEG Audio

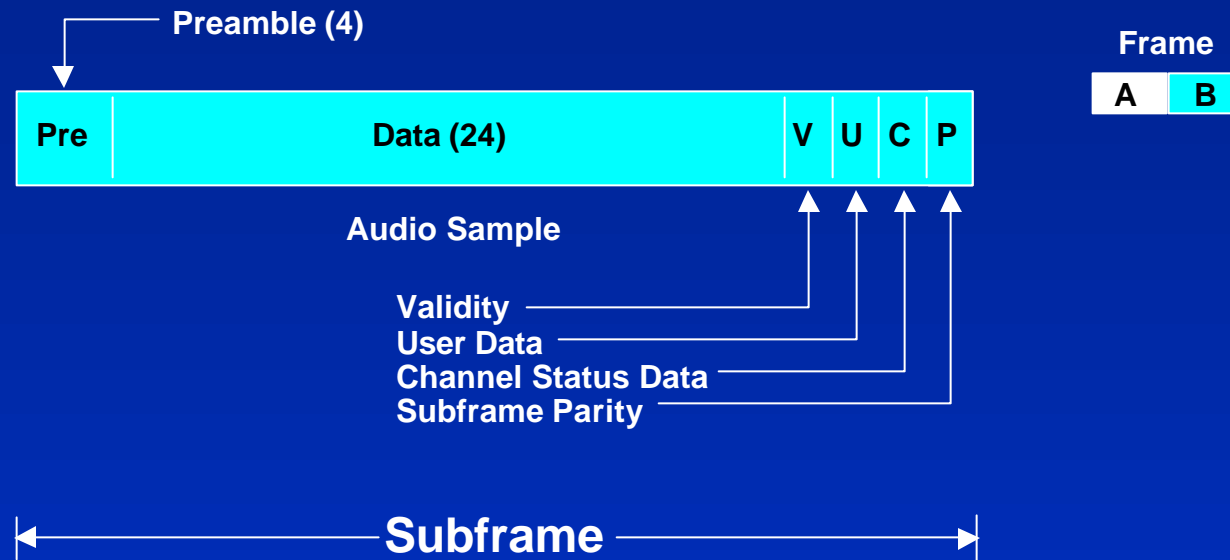
- AES/EBU Audio
- Compression Techniques
- Data structures

AES Digital Audio Signal



- **Type of Manchester code - bi-phase mark coding**
- **Basic characteristics**
- **No dc content**
- **Information contained in transitions**
- **Immune to polarity reversal**
- **Clock signal embedded**

AES/EBU Frames & Subframes



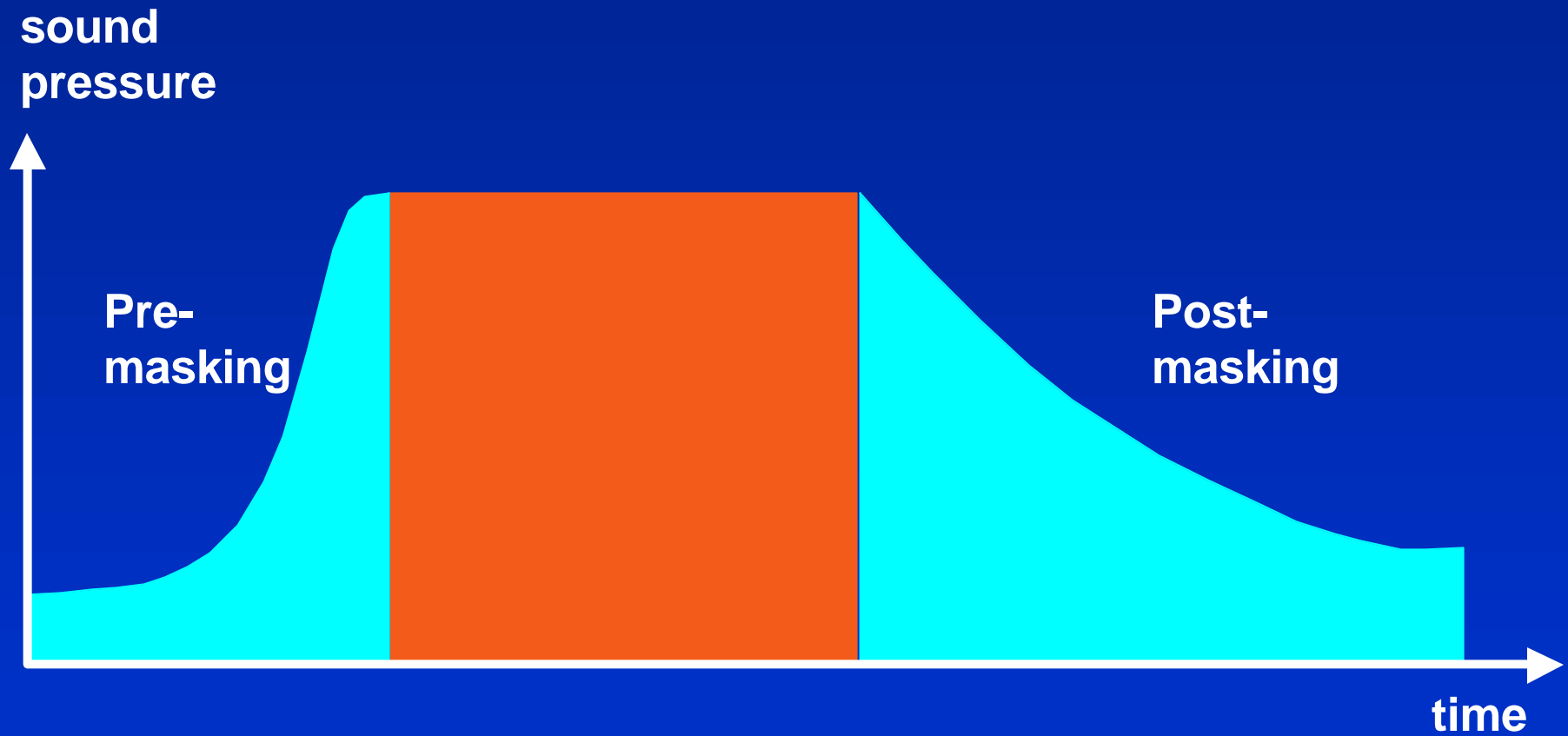
- 32 Bits/subframe
- 4 Preamble Bits, 24 Data Bits, 1 Validity,
- 1 User, 1 Channel Status & 1 Parity

Psychoacoustic Models

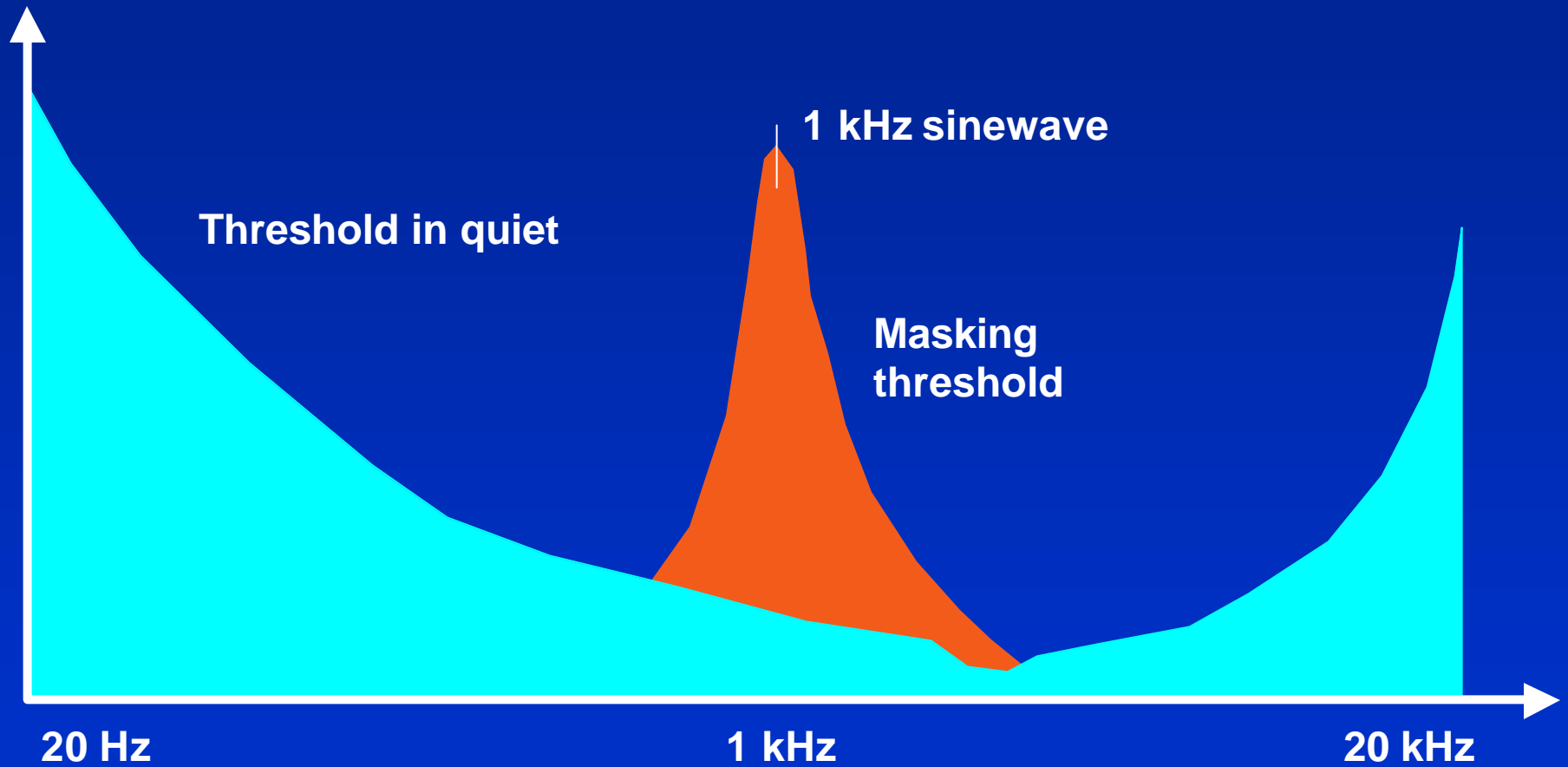
- Pre-Masking
- Post-Masking
- Simultaneous Masking

Premise:
If you can't hear it, don't send it.

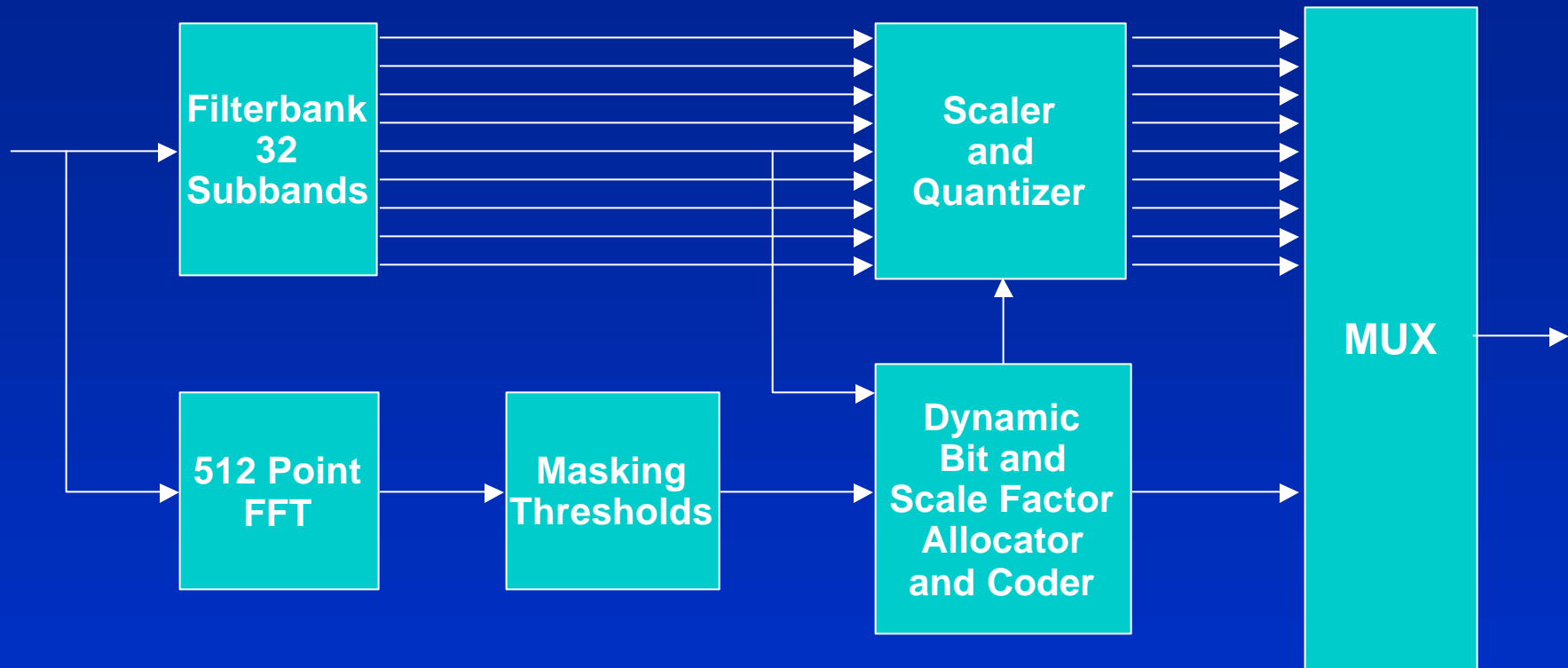
Temporal Masking



Simultaneous Masking

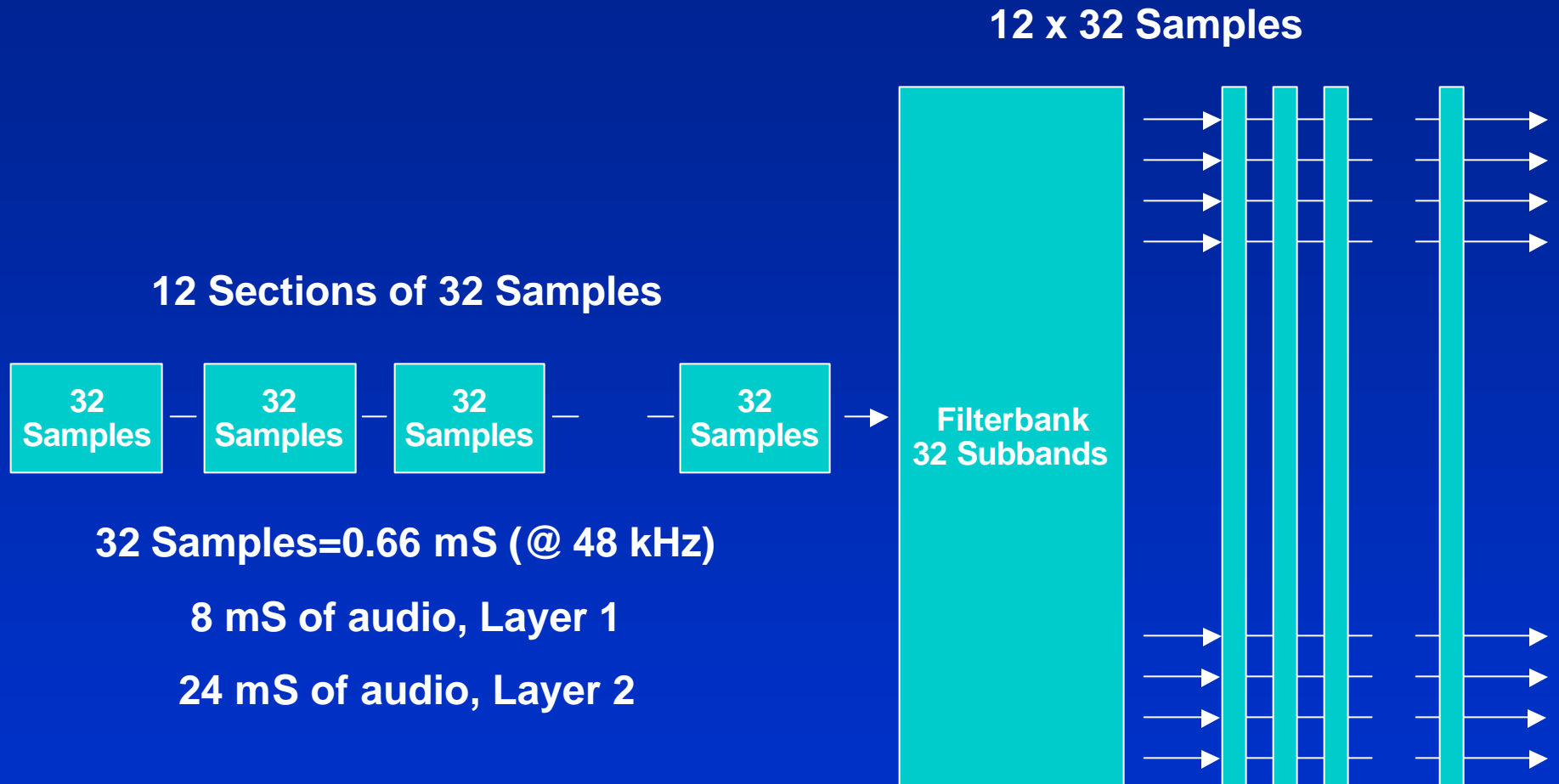


MPEG Audio Encoder



Audio Frame input PCM samples
384 for Layer 1
 $3 * 384 = 1152$ for Layer 2

Audio Time Frame



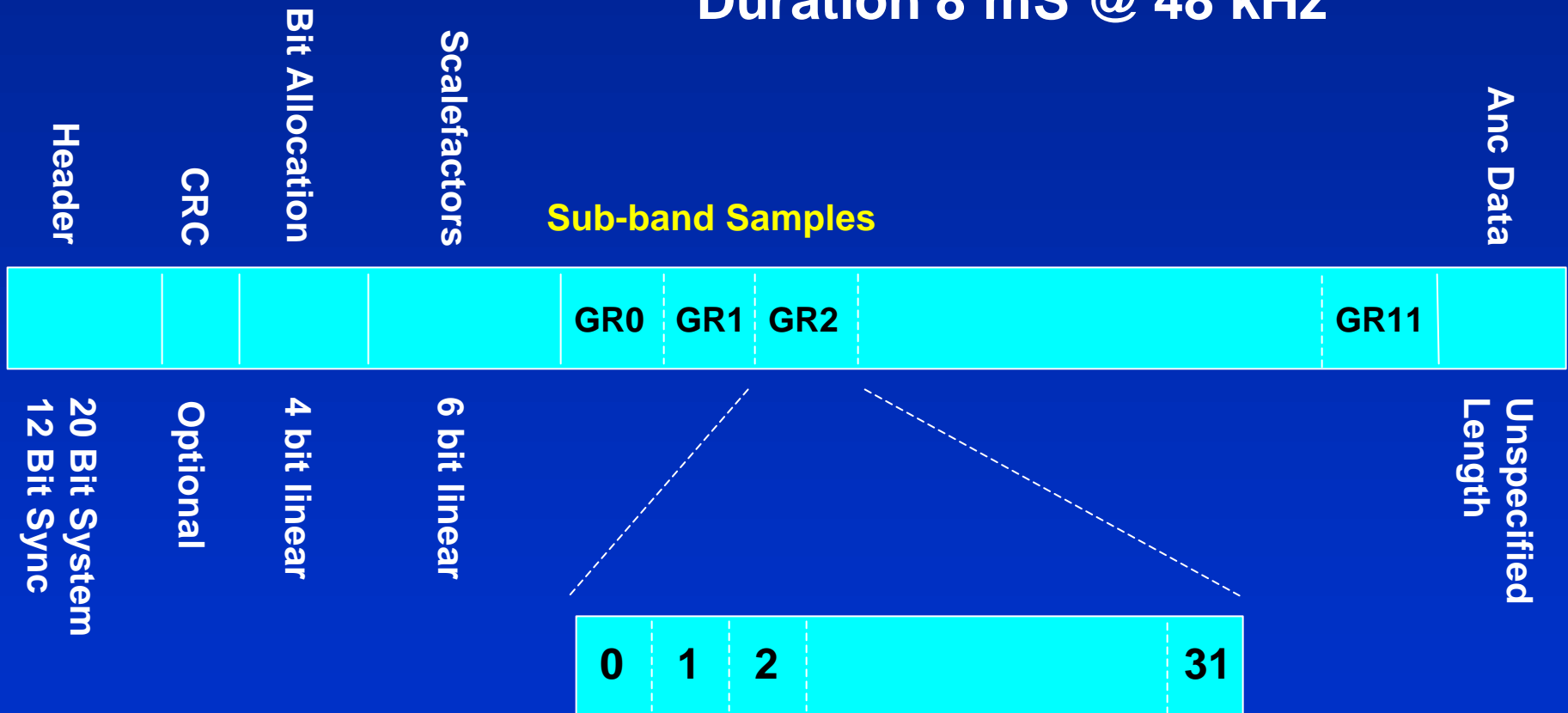
32 Samples = 0.66 mS (@ 48 kHz)

8 mS of audio, Layer 1

24 mS of audio, Layer 2

Layer I Frame Structure

384 PCM Audio Input Samples
Duration 8 mS @ 48 kHz

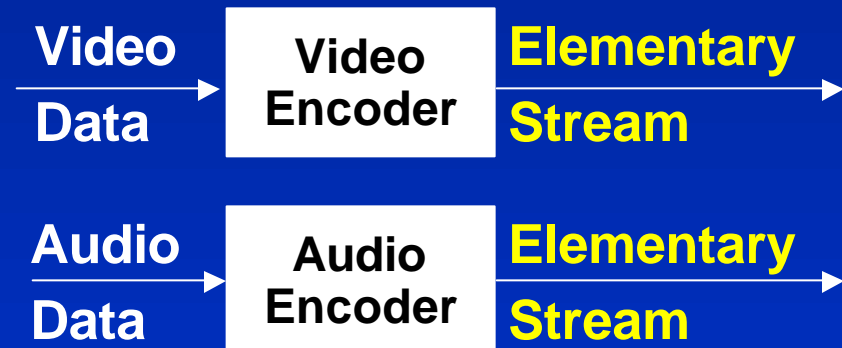


MPEG-2

- **MPEG-2 ISO/IEC 13818/Recommendation H.222.0 (1994)**
 - 13818-1 system level coding (audio and video multiplex)
 - 13818-2 coding and decoding of video data
 - 13818-3 coding of audio data
 - 13818-4 compliance testing of the other three
 - Standard covers the decoder methodology and transport stream syntax

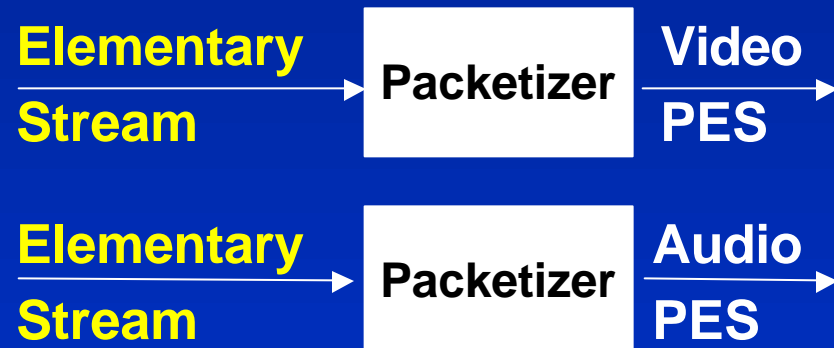
MPEG-2 Encoder/Compressor

- MPEG-2 requires component digital video (Rec. 601).
- The MPEG-2 encoder/ compression product *can* include composite decoders and/or A to D converters.
- The audio input must be AES/EBU digital or the encoder must include a converter/formatter.
- Elementary Streams can be of any length.



MPEG-2 Packetizer

- The Packetizer forms the Video and/or Audio into Packetized Elementary Stream (PES) packets.
- PES Packets contain:
 - Header data
 - Elementary stream data



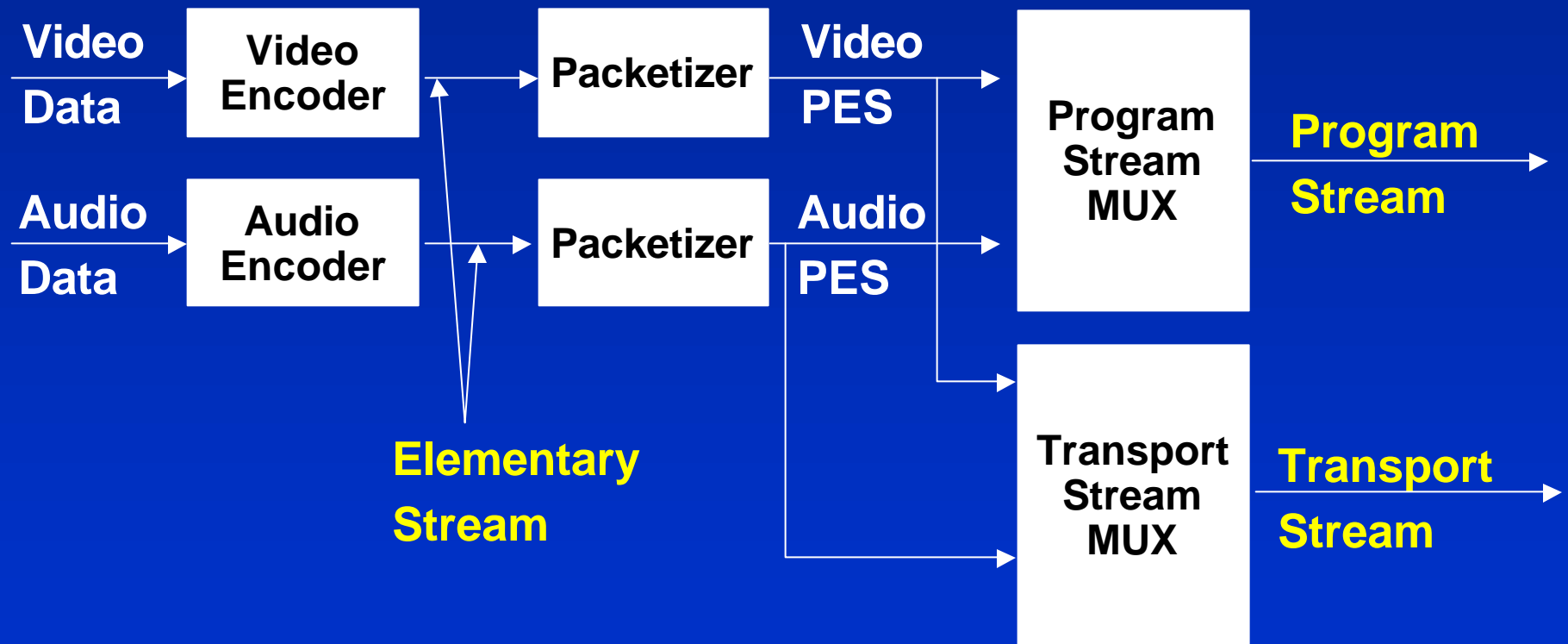
Program Stream Mux

- Combines Video and Audio PES packets into a stream.
- Designed for transmission in error and noise free environments.



MPEG-2 Compression Data

Basic MPEG-2 data layers and terminology

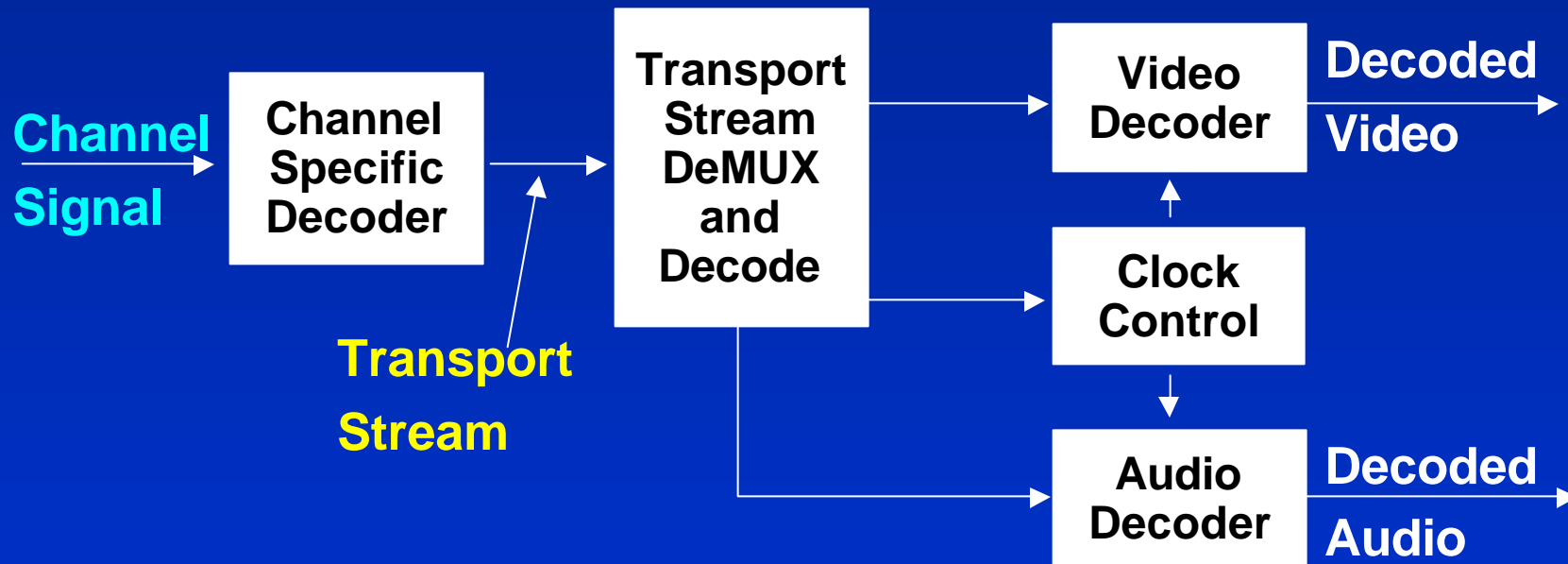


Transport Stream MUX

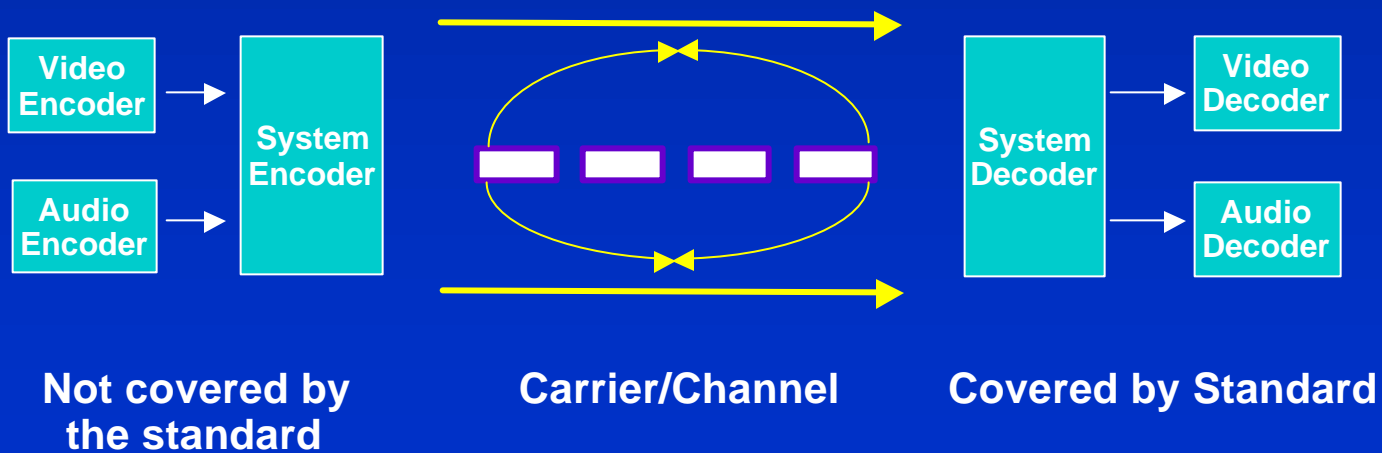
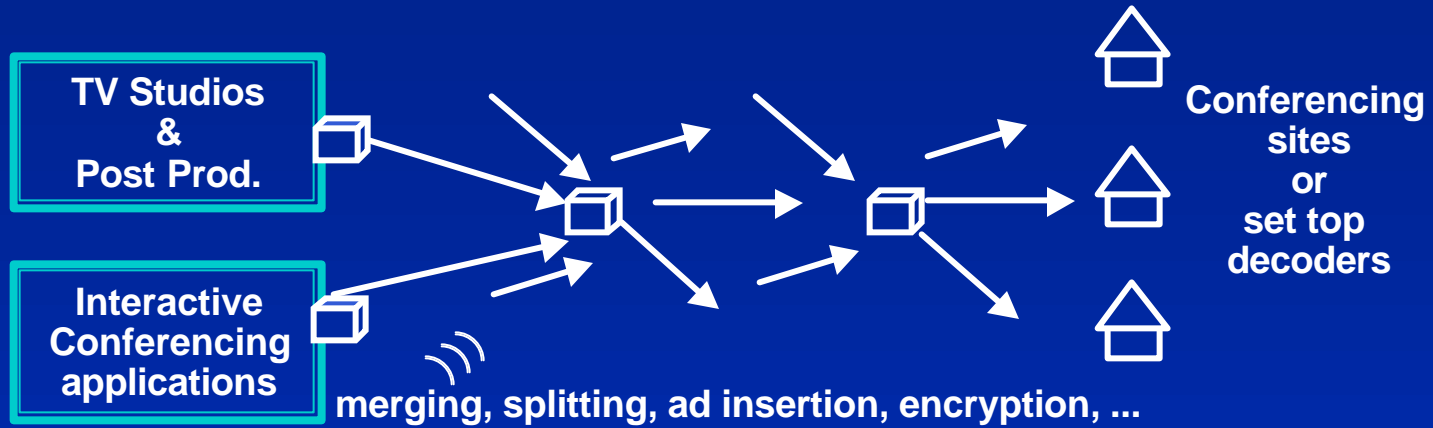
- Combines Video and Audio PES packets into a stream.
- Designed for transmission to ITU-T Rec. H.262, ISO/IEC 13818 standards, and anywhere significant errors may occur.
- The TS contains no error protection itself.
- The use of small packets provides some resistance to the results of errors.
- Combines asynchronous signals to synchronous data stream.



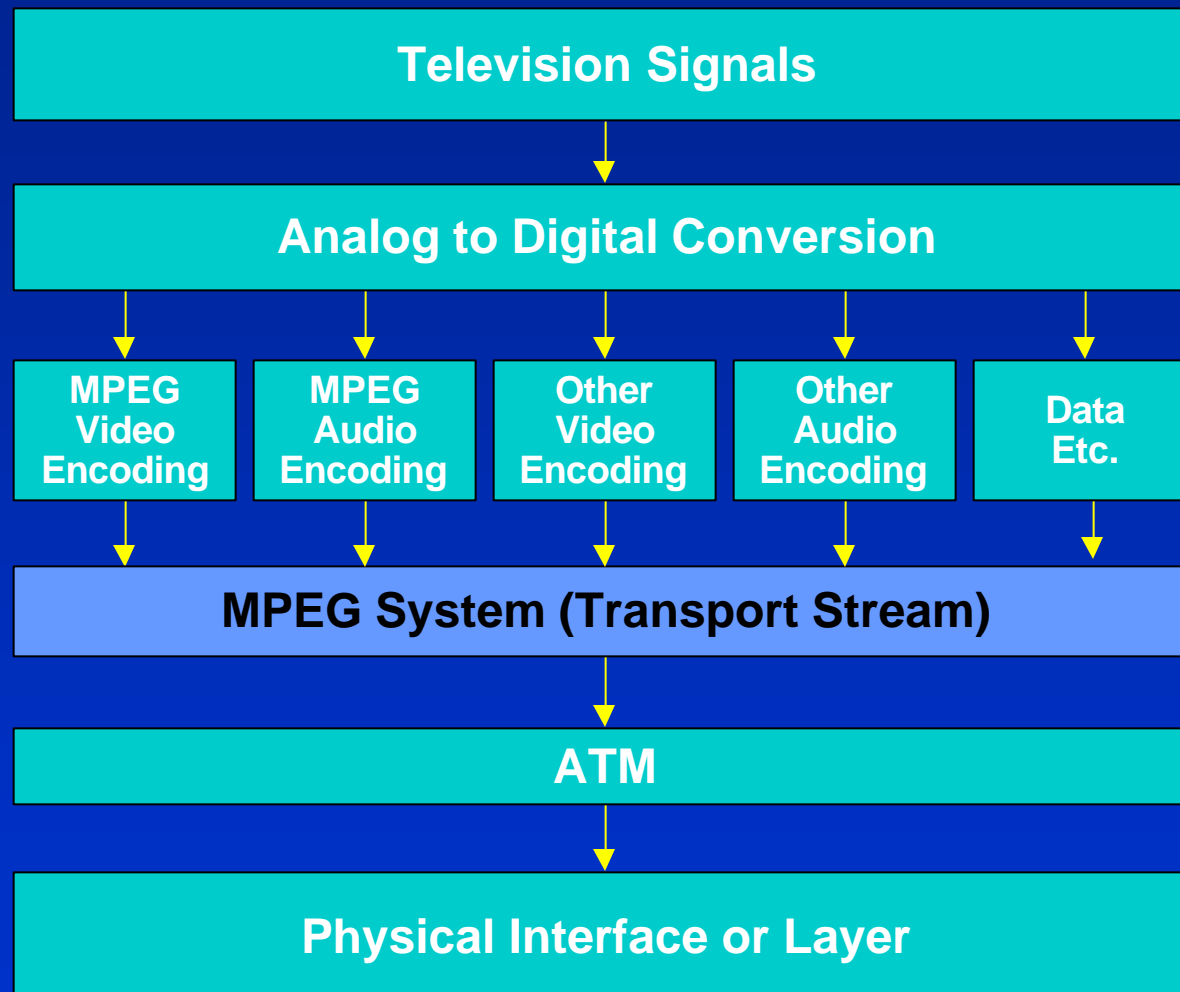
MPEG-2 Decoder



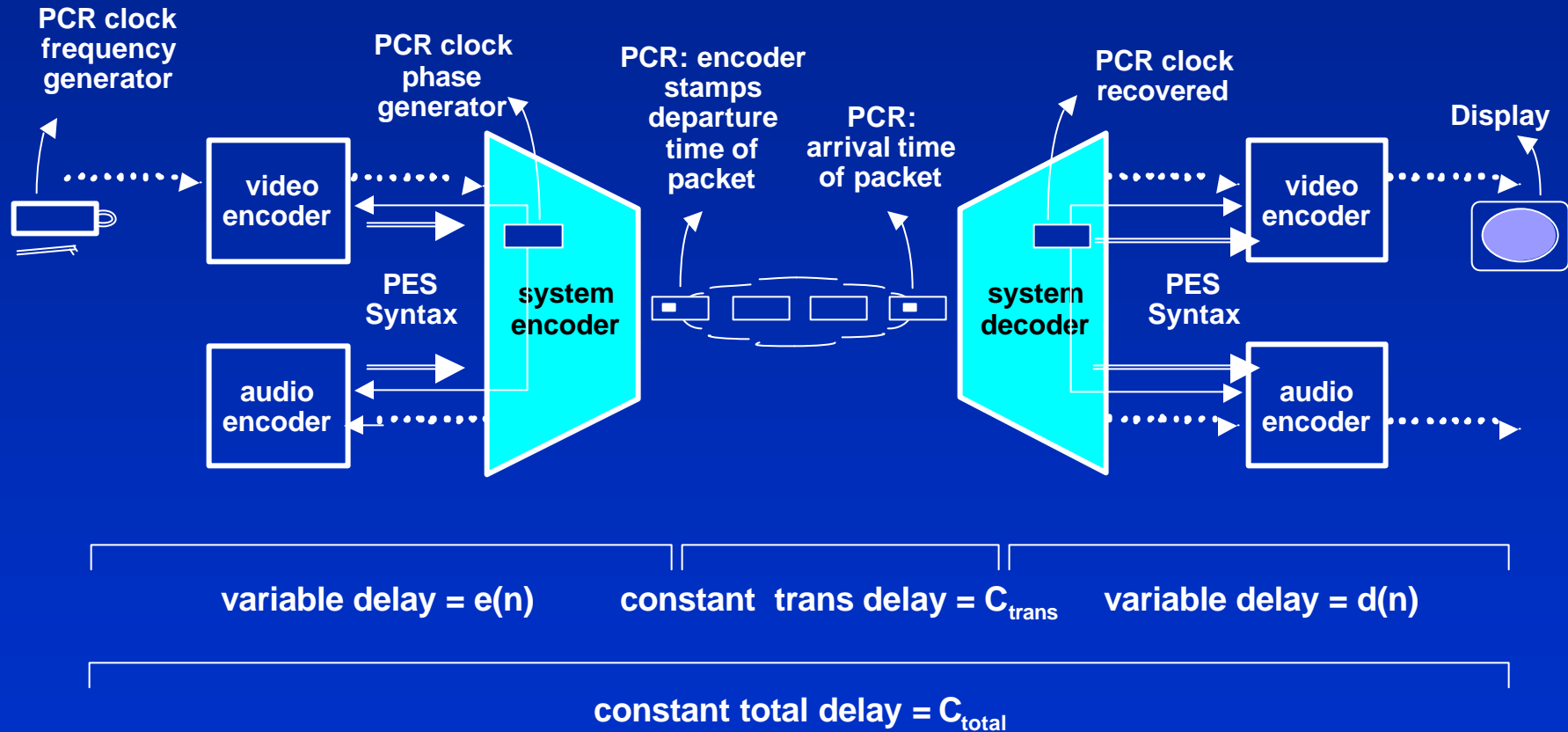
The MPEG-2 System



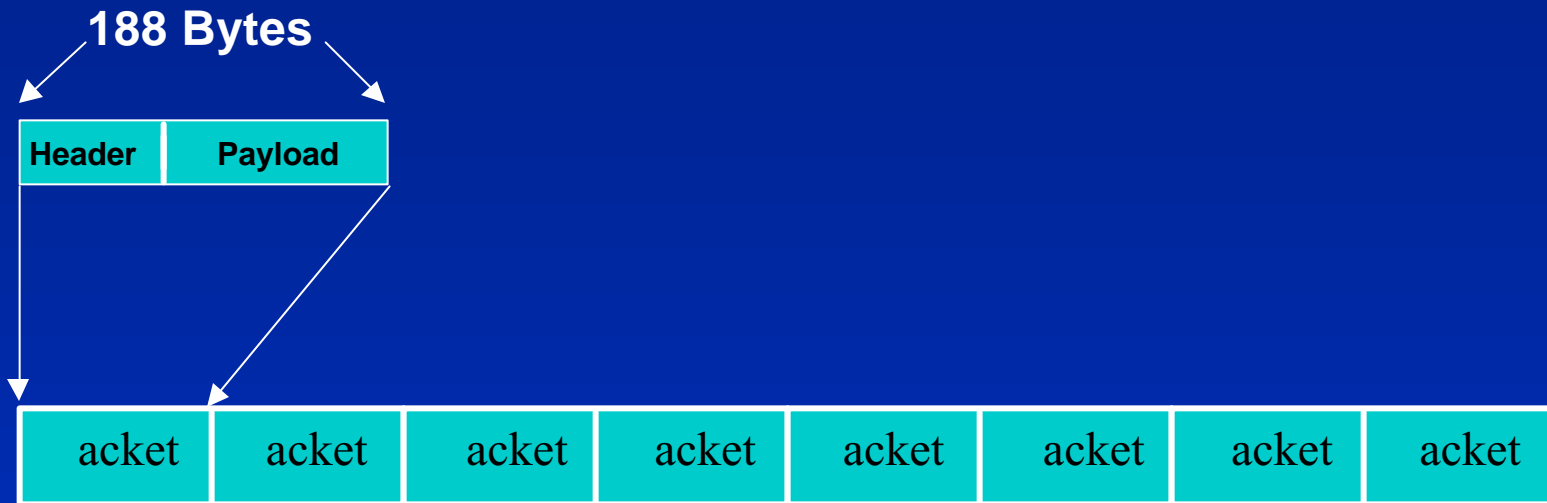
Functional Layers Typical in MPEG



Program Clock Model

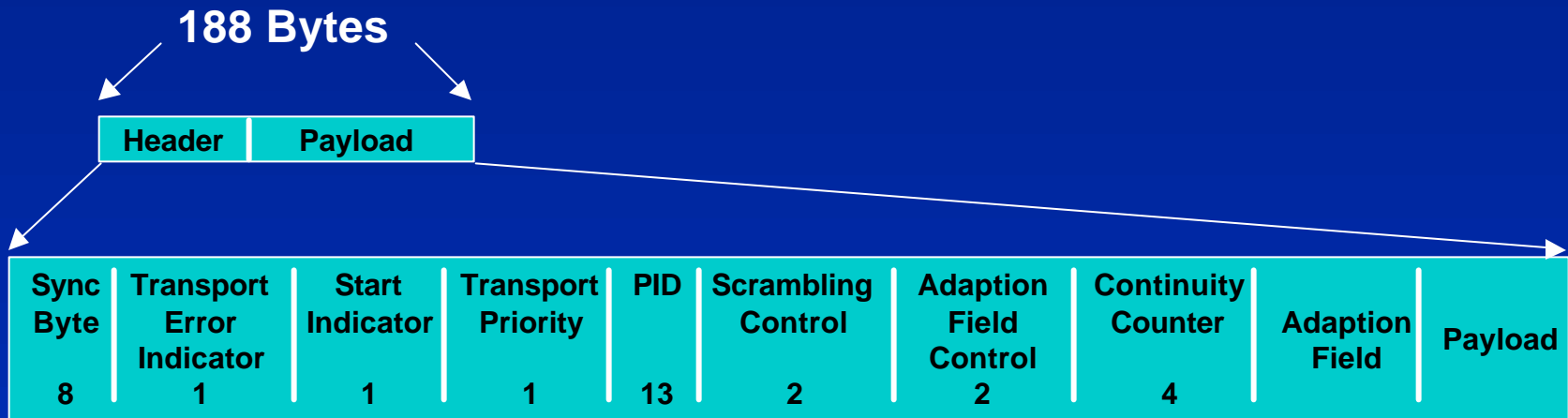


Transport Layer



- **The Transport Stream (TS) is a continuous data stream in 188 byte packets containing format (syntax) information and payload data**

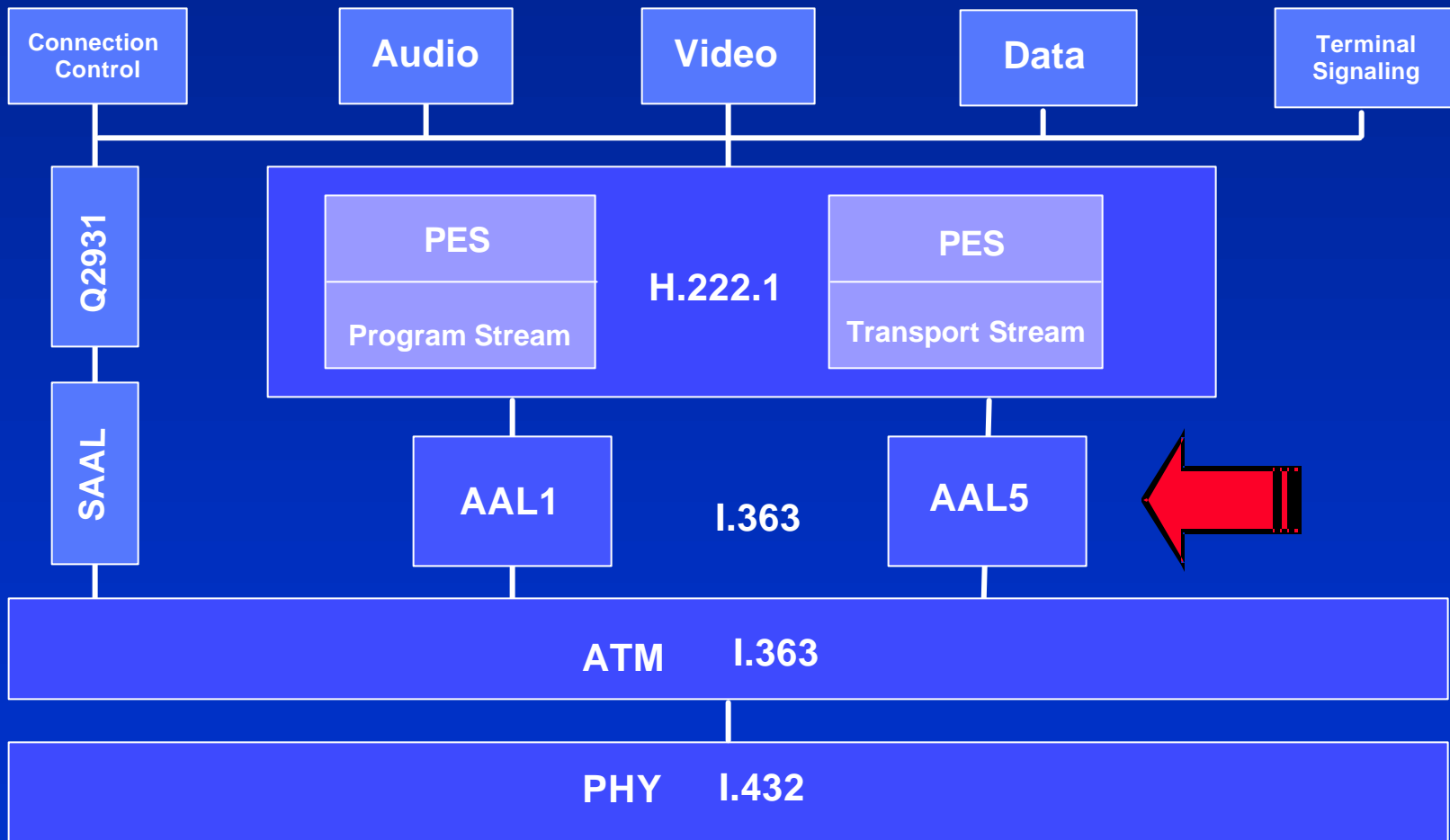
Transport Layer



■ Transport Stream PSI

- Program Association Table (PAT)
- Program Map Tables (PMT)
- Conditional Access Table (CAT)
- Packet Identification (PID)

ATM Multimedia Standards Framework





Information Technology Service : Network settings: quick reference

TCP/IP Settings

IP Address:

- Set your computer to use DHCP (Obtain IP address automatically)

DNS/Name Servers:

- 129.234.4.13
- 129.234.4.11

WINS Servers:

- 129.234.4.60
- 129.234.2.153

Domain Name:

- dur.ac.uk

Hostname:

- Supplied during registration

WebCache/Proxy Settings

Automatic Configuration Script:

- <http://www.dur.ac.uk/Admin/proxy.config>

Manual Proxy Configuration:

- HTTP - `wwwcache.dur.ac.uk:8080`

- HTTPS - `wwwcache.dur.ac.uk:8080`
- FTP - `wwwcache.dur.ac.uk:8080`
- Bypass proxy for local addresses.
- Do not use proxy server for: `www.dur.ac.uk`

Mail and News Server Settings

Incoming Mail Servers

- POP3 - `pop3host.dur.ac.uk`
- IMAP - `imaphost.dur.ac.uk`
- For IMAP Mail - set the IMAP Server/Root Directory to: `~/netscape/mail/`

Outgoing Mail Server:

- SMTP - `smtphost.dur.ac.uk`

News Server

- NNTP - `nntpghost.dur.ac.uk`

LDAP Directory

- LDAP Server: `ldaphost.dur.ac.uk`
- Search String: `o=University of Durham, c=GB`

University of Durham > Information Technology Service > Services & Facilities > Network services > Using the Network > Network settings
skip navigation

search
go

Also in Using the Network

- New network point
- Data network charges
- Registering a network name
- Find the network socket on your PC
- How to install a network card
- Network card drivers
- MAC address
- Configure network settings

- Connect to your J drive
- Connect to a network printer

Network services

- **Using the Network**
- EnSuite Online
- Dialup service
- Mobile computing
- CD-ROM service
- Windows 2000 deployment
- NorMAN

Main sections

- Information
- Helpdesk
- Online Utilities
- Training
- Learning Technologies
- Software & datasets
- **Services & Facilities**
- Policy and Regulations

Information Technology Service
University of Durham
Science Laboratories
South Road
Durham
DH1 3LE

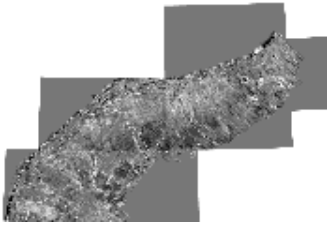
If you want to get in touch with us, please use the Comments and Questions form.

Last Modified: 24 September 2003

Side Scan Sonar

Another use of DSP techniques (including DFT/FFT) is in sonar. The example given here is *side-scan* sonar which is a little different from the normal idea of sonar.

With this method a 6.5-Khz sound pulse is transmitted into the ocean toward the sea floor at an oblique angle. Because the signal is transmitted at an oblique angle rather than straight down, the reflected signal provides information about the inclination of the sea floor, surface roughness, and acoustic impedance of the sea floor. It also highlights any small structures on the sea floor, folds, and any fault lines that may be present.



This image is a composite of 2 degree by 2 degree maps covering the northern east coast of the United States.

The above image came from the United States Geological Survey (USGS) which is using a physical Long Range ASDIC (GLORIA) sidescan sonar system to obtain a plan view of the sea floor of the Exclusive Economic Zone (EEZ). The picture element (pixel) resolution is approximately 50 meters. The data are digitally mosaicked into image maps which are at a scale of 1:500,000. These mosaics provide the equivalent of "aerial photographs" that reveal many physiographic and geologic features of the sea floor.

To date the project has covered approximately 2 million square nautical miles of sea floor seaward of the shelf edge around the 30 coastal States. Mapping is continuing around the American Flag Islands of the central and western Pacific Ocean.

World Wide Web

The United States Geological Survey (USGS) has further data available via the World Wide Web.

Back to *List of other applications*



INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

T.81

(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

	<i>Page</i>
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 requirements 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 developed a compression standard to meet the needs of other applications as well, including desktop publishing, graphic arts, medical imaging and scientific imaging.

Annexes A, B, C, D, E, F, G, H and J are normative, and thus form an integral part of this Specification. Annexes K, L and M are informative and thus do not form an integral part of this Specification.

This Specification aims to follow the guidelines of CCITT and ISO/IEC JTC 1 on *Rules for presentation of CCITT | ISO/IEC common text*.

INTERNATIONAL STANDARD

CCITT RECOMMENDATION

**INFORMATION TECHNOLOGY – DIGITAL COMPRESSION
AND CODING OF CONTINUOUS-TONE STILL IMAGES –
REQUIREMENTS AND GUIDELINES**

1 Scope

This CCITT Recommendation | International Standard is applicable to continuous-tone – grayscale or colour – digital still image data. It is applicable to a wide range of applications which require use of compressed images. It is not applicable to bi-level image data.

This Specification

- specifies processes for converting source image data to compressed image data;
- specifies processes for converting compressed image data to reconstructed image data;
- gives guidance on how to implement these processes in practice;
- specifies coded representations for compressed image data.

NOTE – This Specification does not specify a complete coded image representation. Such representations may include certain parameters, such as aspect ratio, component sample registration, and colour space designation, which are application-dependent.

2 Normative references

The following CCITT Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this CCITT Recommendation | International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this CCITT Recommendation | International Standard are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The CCITT Secretariat maintains a list of currently valid CCITT Recommendations.

- *CCITT Recommendation T.80 (1992), Common components for image compression and communication – Basic principles.*

3 Definitions, abbreviations and symbols**3.1 Definitions and abbreviations**

For the purposes of this Specification, the following definitions apply.

3.1.1 abbreviated format: A representation of compressed image data which is missing some or all of the table specifications required for decoding, or a representation of table-specification data without frame headers, scan headers, and entropy-coded segments.

3.1.2 AC coefficient: Any DCT coefficient for which the frequency is not zero in at least one dimension.

3.1.3 (adaptive) (binary) arithmetic decoding: An entropy decoding procedure which recovers the sequence of symbols from the sequence of bits produced by the arithmetic encoder.

3.1.4 (adaptive) (binary) arithmetic encoding: An entropy encoding procedure which codes by means of a recursive subdivision of the probability of the sequence of symbols coded up to that point.

3.1.5 application environment: The standards for data representation, communication, or storage which have been established for a particular application.

- 3.1.6 **arithmetic decoder:** An embodiment of arithmetic decoding procedure.
- 3.1.7 **arithmetic encoder:** An embodiment of arithmetic encoding procedure.
- 3.1.8 **baseline (sequential):** A particular sequential DCT-based encoding and decoding process specified in this Specification, and which is required for all DCT-based decoding processes.
- 3.1.9 **binary decision:** Choice between two alternatives.
- 3.1.10 **bit stream:** Partially encoded or decoded sequence of bits comprising an entropy-coded segment.
- 3.1.11 **block:** An 8×8 array of samples or an 8×8 array of DCT coefficient values of one component.
- 3.1.12 **block-row:** A sequence of eight contiguous component lines which are partitioned into 8×8 blocks.
- 3.1.13 **byte:** A group of 8 bits.
- 3.1.14 **byte stuffing:** A procedure in which either the Huffman coder or the arithmetic coder inserts a zero byte into the entropy-coded segment following the generation of an encoded hexadecimal X'FF' byte.
- 3.1.15 **carry bit:** A bit in the arithmetic encoder code register which is set if a carry-over in the code register overflows the eight bits reserved for the output byte.
- 3.1.16 **ceiling function:** The mathematical procedure in which the greatest integer value of a real number is obtained by selecting the smallest integer value which is greater than or equal to the real number.
- 3.1.17 **class (of coding process):** Lossy or lossless coding processes.
- 3.1.18 **code register:** The arithmetic encoder register containing the least significant bits of the partially completed entropy-coded segment. Alternatively, the arithmetic decoder register containing the most significant bits of a partially decoded entropy-coded segment.
- 3.1.19 **coder:** An embodiment of a coding process.
- 3.1.20 **coding:** Encoding or decoding.
- 3.1.21 **coding model:** A procedure used to convert input data into symbols to be coded.
- 3.1.22 **(coding) process:** A general term for referring to an encoding process, a decoding process, or both.
- 3.1.23 **colour image:** A continuous-tone image that has more than one component.
- 3.1.24 **columns:** Samples per line in a component.
- 3.1.25 **component:** One of the two-dimensional arrays which comprise an image.
- 3.1.26 **compressed data:** Either compressed image data or table specification data or both.
- 3.1.27 **compressed image data:** A coded representation of an image, as specified in this Specification.
- 3.1.28 **compression:** Reduction in the number of bits used to represent source image data.
- 3.1.29 **conditional exchange:** The interchange of MPS and LPS probability intervals whenever the size of the LPS interval is greater than the size of the MPS interval (in arithmetic coding).
- 3.1.30 **(conditional) probability estimate:** The probability value assigned to the LPS by the probability estimation state machine (in arithmetic coding).
- 3.1.31 **conditioning table:** The set of parameters which select one of the defined relationships between prior coding decisions and the conditional probability estimates used in arithmetic coding.
- 3.1.32 **context:** The set of previously coded binary decisions which is used to create the index to the probability estimation state machine (in arithmetic coding).
- 3.1.33 **continuous-tone image:** An image whose components have more than one bit per sample.
- 3.1.34 **data unit:** An 8×8 block of samples of one component in DCT-based processes; a sample in lossless processes.

- 3.1.35 DC coefficient:** The DCT coefficient for which the frequency is zero in both dimensions.
- 3.1.36 DC prediction:** The procedure used by DCT-based encoders whereby the quantized DC coefficient from the previously encoded 8×8 block of the same component is subtracted from the current quantized DC coefficient.
- 3.1.37 (DCT) coefficient:** The amplitude of a specific cosine basis function – may refer to an original DCT coefficient, to a quantized DCT coefficient, or to a dequantized DCT coefficient.
- 3.1.38 decoder:** An embodiment of a decoding process.
- 3.1.39 decoding process:** A process which takes as its input compressed image data and outputs a continuous-tone image.
- 3.1.40 default conditioning:** The values defined for the arithmetic coding conditioning tables at the beginning of coding of an image.
- 3.1.41 dequantization:** The inverse procedure to quantization by which the decoder recovers a representation of the DCT coefficients.
- 3.1.42 differential component:** The difference between an input component derived from the source image and the corresponding reference component derived from the preceding frame for that component (in hierarchical mode coding).
- 3.1.43 differential frame:** A frame in a hierarchical process in which differential components are either encoded or decoded.
- 3.1.44 (digital) reconstructed image (data):** A continuous-tone image which is the output of any decoder defined in this Specification.
- 3.1.45 (digital) source image (data):** A continuous-tone image used as input to any encoder defined in this Specification.
- 3.1.46 (digital) (still) image:** A set of two-dimensional arrays of integer data.
- 3.1.47 discrete cosine transform; DCT:** Either the forward discrete cosine transform or the inverse discrete cosine transform.
- 3.1.48 downsampling (filter):** A procedure by which the spatial resolution of an image is reduced (in hierarchical mode coding).
- 3.1.49 encoder:** An embodiment of an encoding process.
- 3.1.50 encoding process:** A process which takes as its input a continuous-tone image and outputs compressed image data.
- 3.1.51 entropy-coded (data) segment:** An independently decodable sequence of entropy encoded bytes of compressed image data.
- 3.1.52 (entropy-coded segment) pointer:** The variable which points to the most recently placed (or fetched) byte in the entropy encoded segment.
- 3.1.53 entropy decoder:** An embodiment of an entropy decoding procedure.
- 3.1.54 entropy decoding:** A lossless procedure which recovers the sequence of symbols from the sequence of bits produced by the entropy encoder.
- 3.1.55 entropy encoder:** An embodiment of an entropy encoding procedure.
- 3.1.56 entropy encoding:** A lossless procedure which converts a sequence of input symbols into a sequence of bits such that the average number of bits per symbol approaches the entropy of the input symbols.
- 3.1.57 extended (DCT-based) process:** A descriptive term for DCT-based encoding and decoding processes in which additional capabilities are added to the baseline sequential process.
- 3.1.58 forward discrete cosine transform; FDCT:** A mathematical transformation using cosine basis functions which converts a block of samples into a corresponding block of original DCT coefficients.

- 3.1.59 frame:** A group of one or more scans (all using the same DCT-based or lossless process) through the data of one or more of the components in an image.
- 3.1.60 frame header:** A marker segment that contains a start-of-frame marker and associated frame parameters that are coded at the beginning of a frame.
- 3.1.61 frequency:** A two-dimensional index into the two-dimensional array of DCT coefficients.
- 3.1.62 (frequency) band:** A contiguous group of coefficients from the zig-zag sequence (in progressive mode coding).
- 3.1.63 full progression:** A process which uses both spectral selection and successive approximation (in progressive mode coding).
- 3.1.64 grayscale image:** A continuous-tone image that has only one component.
- 3.1.65 hierarchical:** A mode of operation for coding an image in which the first frame for a given component is followed by frames which code the differences between the source data and the reconstructed data from the previous frame for that component. Resolution changes are allowed between frames.
- 3.1.66 hierarchical decoder:** A sequence of decoder processes in which the first frame for each component is followed by frames which decode an array of differences for each component and adds it to the reconstructed data from the preceding frame for that component.
- 3.1.67 hierarchical encoder:** The mode of operation in which the first frame for each component is followed by frames which encode the array of differences between the source data and the reconstructed data from the preceding frame for that component.
- 3.1.68 horizontal sampling factor:** The relative number of horizontal data units of a particular component with respect to the number of horizontal data units in the other components.
- 3.1.69 Huffman decoder:** An embodiment of a Huffman decoding procedure.
- 3.1.70 Huffman decoding:** An entropy decoding procedure which recovers the symbol from each variable length code produced by the Huffman encoder.
- 3.1.71 Huffman encoder:** An embodiment of a Huffman encoding procedure.
- 3.1.72 Huffman encoding:** An entropy encoding procedure which assigns a variable length code to each input symbol.
- 3.1.73 Huffman table:** The set of variable length codes required in a Huffman encoder and Huffman decoder.
- 3.1.74 image data:** Either source image data or reconstructed image data.
- 3.1.75 interchange format:** The representation of compressed image data for exchange between application environments.
- 3.1.76 interleaved:** The descriptive term applied to the repetitive multiplexing of small groups of data units from each component in a scan in a specific order.
- 3.1.77 inverse discrete cosine transform; IDCT:** A mathematical transformation using cosine basis functions which converts a block of dequantized DCT coefficients into a corresponding block of samples.
- 3.1.78 Joint Photographic Experts Group; JPEG:** The informal name of the committee which created this Specification. The "joint" comes from the CCITT and ISO/IEC collaboration.
- 3.1.79 latent output:** Output of the arithmetic encoder which is held, pending resolution of carry-over (in arithmetic coding).
- 3.1.80 less probable symbol; LPS:** For a binary decision, the decision value which has the smaller probability.
- 3.1.81 level shift:** A procedure used by DCT-based encoders and decoders whereby each input sample is either converted from an unsigned representation to a two's complement representation or from a two's complement representation to an unsigned representation.

- 3.1.82 lossless:** A descriptive term for encoding and decoding processes and procedures in which the output of the decoding procedure(s) is identical to the input to the encoding procedure(s).
- 3.1.83 lossless coding:** The mode of operation which refers to any one of the coding processes defined in this Specification in which all of the procedures are lossless (see Annex H).
- 3.1.84 lossy:** A descriptive term for encoding and decoding processes which are not lossless.
- 3.1.85 marker:** A two-byte code in which the first byte is hexadecimal FF (X'FF') and the second byte is a value between 1 and hexadecimal FE (X'FE').
- 3.1.86 marker segment:** A marker and associated set of parameters.
- 3.1.87 MCU-row:** The smallest sequence of MCU which contains at least one line of samples or one block-row from every component in the scan.
- 3.1.88 minimum coded unit; MCU:** The smallest group of data units that is coded.
- 3.1.89 modes (of operation):** The four main categories of image coding processes defined in this Specification.
- 3.1.90 more probable symbol; MPS:** For a binary decision, the decision value which has the larger probability.
- 3.1.91 non-differential frame:** The first frame for any components in a hierarchical encoder or decoder. The components are encoded or decoded without subtraction from reference components. The term refers also to any frame in modes other than the hierarchical mode.
- 3.1.92 non-interleaved:** The descriptive term applied to the data unit processing sequence when the scan has only one component.
- 3.1.93 parameters:** Fixed length integers 4, 8 or 16 bits in length, used in the compressed data formats.
- 3.1.94 point transform:** Scaling of a sample or DCT coefficient.
- 3.1.95 precision:** Number of bits allocated to a particular sample or DCT coefficient.
- 3.1.96 predictor:** A linear combination of previously reconstructed values (in lossless mode coding).
- 3.1.97 probability estimation state machine:** An interlinked table of probability values and indices which is used to estimate the probability of the LPS (in arithmetic coding).
- 3.1.98 probability interval:** The probability of a particular sequence of binary decisions within the ordered set of all possible sequences (in arithmetic coding).
- 3.1.99 (probability) sub-interval:** A portion of a probability interval allocated to either of the two possible binary decision values (in arithmetic coding).
- 3.1.100 procedure:** A set of steps which accomplishes one of the tasks which comprise an encoding or decoding process.
- 3.1.101 process:** See coding process.
- 3.1.102 progressive (coding):** One of the DCT-based processes defined in this Specification in which each scan typically improves the quality of the reconstructed image.
- 3.1.103 progressive DCT-based:** The mode of operation which refers to any one of the processes defined in Annex G.
- 3.1.104 quantization table:** The set of 64 quantization values used to quantize the DCT coefficients.
- 3.1.105 quantization value:** An integer value used in the quantization procedure.
- 3.1.106 quantize:** The act of performing the quantization procedure for a DCT coefficient.
- 3.1.107 reference (reconstructed) component:** Reconstructed component data which is used in a subsequent frame of a hierarchical encoder or decoder process (in hierarchical mode coding).

- 3.1.108 renormalization:** The doubling of the probability interval and the code register value until the probability interval exceeds a fixed minimum value (in arithmetic coding).
- 3.1.109 restart interval:** The integer number of MCUs processed as an independent sequence within a scan.
- 3.1.110 restart marker:** The marker that separates two restart intervals in a scan.
- 3.1.111 run (length):** Number of consecutive symbols of the same value.
- 3.1.112 sample:** One element in the two-dimensional array which comprises a component.
- 3.1.113 sample-interleaved:** The descriptive term applied to the repetitive multiplexing of small groups of samples from each component in a scan in a specific order.
- 3.1.114 scan:** A single pass through the data for one or more of the components in an image.
- 3.1.115 scan header:** A marker segment that contains a start-of-scan marker and associated scan parameters that are coded at the beginning of a scan.
- 3.1.116 sequential (coding):** One of the lossless or DCT-based coding processes defined in this Specification in which each component of the image is encoded within a single scan.
- 3.1.117 sequential DCT-based:** The mode of operation which refers to any one of the processes defined in Annex F.
- 3.1.118 spectral selection:** A progressive coding process in which the zig-zag sequence is divided into bands of one or more contiguous coefficients, and each band is coded in one scan.
- 3.1.119 stack counter:** The count of X'FF' bytes which are held, pending resolution of carry-over in the arithmetic encoder.
- 3.1.120 statistical conditioning:** The selection, based on prior coding decisions, of one estimate out of a set of conditional probability estimates (in arithmetic coding).
- 3.1.121 statistical model:** The assignment of a particular conditional probability estimate to each of the binary arithmetic coding decisions.
- 3.1.122 statistics area:** The array of statistics bins required for a coding process which uses arithmetic coding.
- 3.1.123 statistics bin:** The storage location where an index is stored which identifies the value of the conditional probability estimate used for a particular arithmetic coding binary decision.
- 3.1.124 successive approximation:** A progressive coding process in which the coefficients are coded with reduced precision in the first scan, and precision is increased by one bit with each succeeding scan.
- 3.1.125 table specification data:** The coded representation from which the tables used in the encoder and decoder are generated and their destinations specified.
- 3.1.126 transcoder:** A procedure for converting compressed image data of one encoder process to compressed image data of another encoder process.
- 3.1.127 (uniform) quantization:** The procedure by which DCT coefficients are linearly scaled in order to achieve compression.
- 3.1.128 upsampling (filter):** A procedure by which the spatial resolution of an image is increased (in hierarchical mode coding).
- 3.1.129 vertical sampling factor:** The relative number of vertical data units of a particular component with respect to the number of vertical data units in the other components in the frame.
- 3.1.130 zero byte:** The X'00' byte.
- 3.1.131 zig-zag sequence:** A specific sequential ordering of the DCT coefficients from (approximately) lowest spatial frequency to highest.
- 3.1.132 3-sample predictor:** A linear combination of the three nearest neighbor reconstructed samples to the left and above (in lossless mode coding).

3.2 Symbols

The symbols used in this Specification are listed below.

A	probability interval
AC	AC DCT coefficient
AC _{ji}	AC coefficient predicted from DC values
A _h	successive approximation bit position, high
A _l	successive approximation bit position, low
A _{p_i}	<i>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
B _x	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
C _{m_i}	<i>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
C _s	conditioning table value
C _{s_i}	component identifier for scan
CT	renormalization shift counter
C _x	high order 16 bits of arithmetic decoder code register
CX	conditional exchange
d _{ji}	data unit from horizontal position <i>i</i> , vertical position <i>j</i>
d _{ji^k}	d _{ji} for component <i>k</i>
D	decision decoded

ISO/IEC 10918-1 : 1993(E)

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>i</i> th block in component
DC _k	<i>k</i> 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>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
EOB _n	run length category for EOB runs
EOB _x	position of EOB in previous successive approximation scan
EOB ₀ , EOB ₁ , ..., EOB ₁₄	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>i</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>i</i>	subscript index
I	integer variable
Index(S)	index to probability estimation state machine table for context index S
<i>j</i>	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
M _n	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

ISO/IEC 10918-1 : 1993(E)

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	kth quantized DC value used in prediction of AC coefficients
Qe	LPS probability estimate
Qe(S)	LPS probability estimate for context index S
Qk	kth 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

ISO/IEC 10918-1 : 1993(E)

Ta _j	AC entropy table destination selector for <i>j</i> 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 <i>j</i> 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>i</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>i</i> th component
V _{i,j}	<i>j</i> th value for length <i>i</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>i</i> th component
X	number of samples per line in component with largest horizontal dimension
X _i	<i>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>i</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)	<i>K</i> 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:

- 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*.
- 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*.
- 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*.

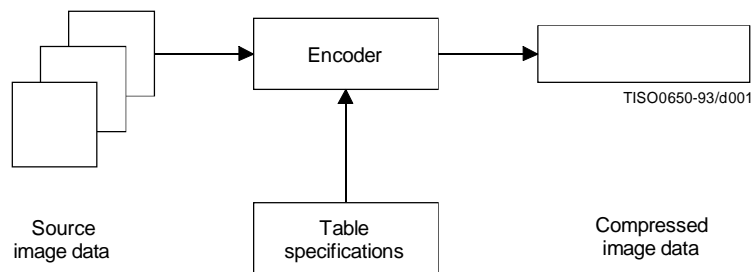


Figure 1 – Encoder

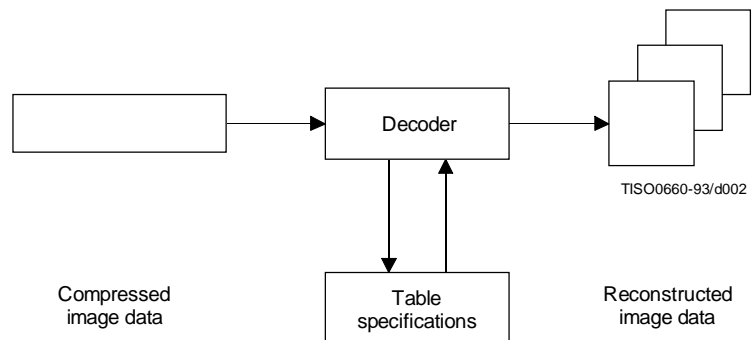


Figure 2 – Decoder

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.

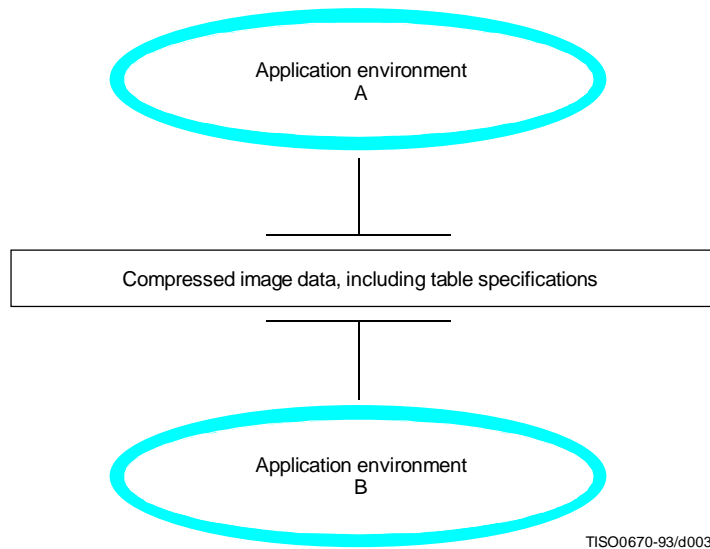


Figure 3 – Interchange format for compressed image data

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.

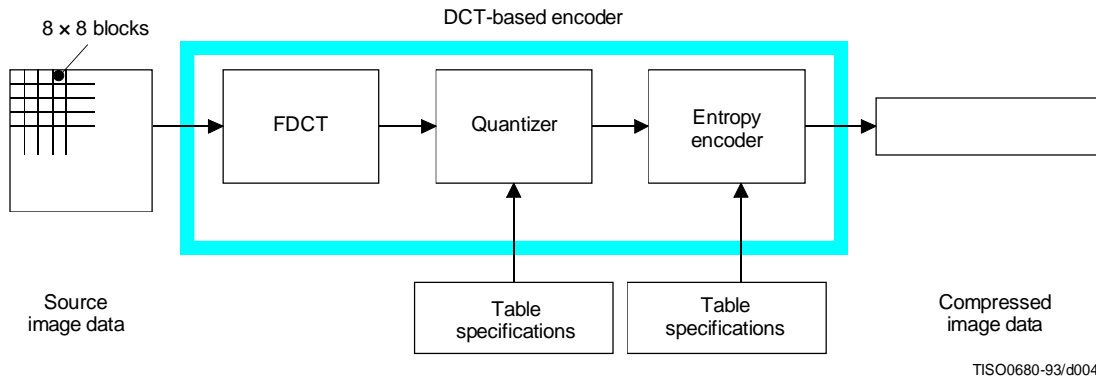


Figure 4 – DCT-based encoder simplified diagram

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.

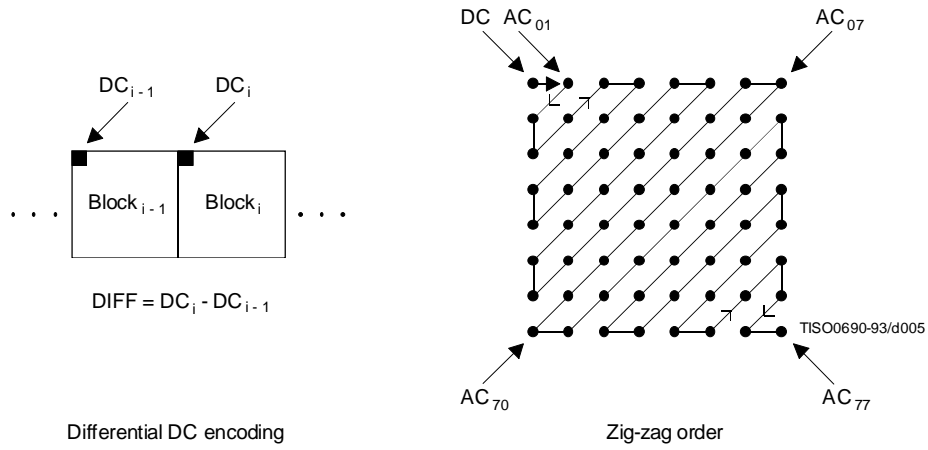


Figure 5 – Preparation of quantized coefficients for entropy encoding

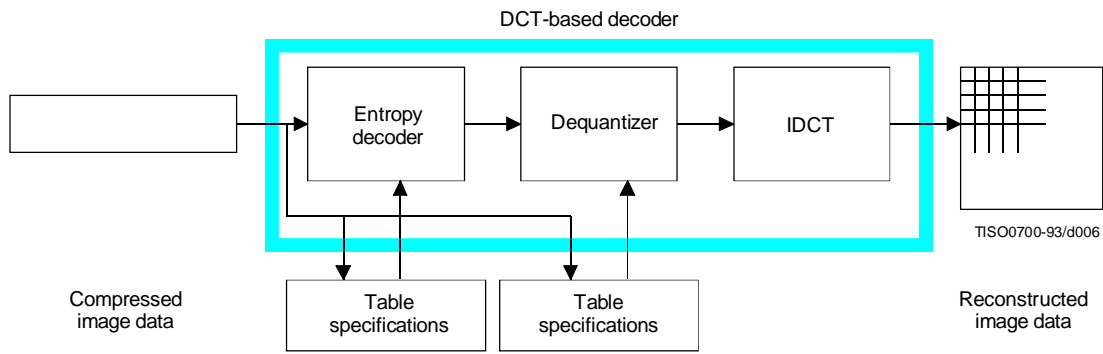


Figure 6 – DCT-based decoder simplified diagram

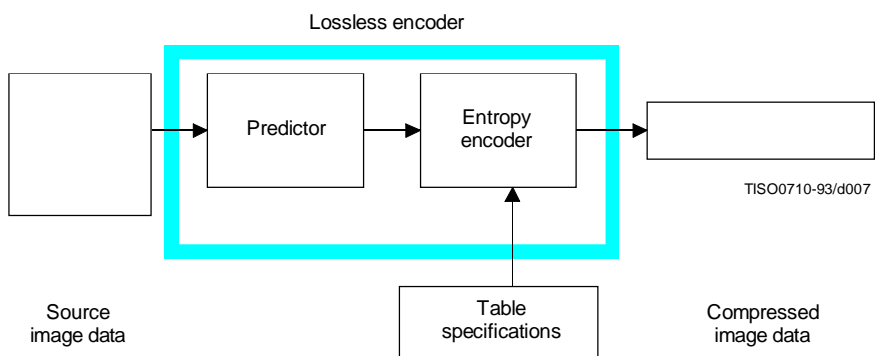


Figure 7 – Lossless encoder simplified diagram

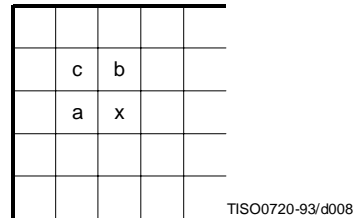


Figure 8 – 3-sample prediction neighbourhood

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.

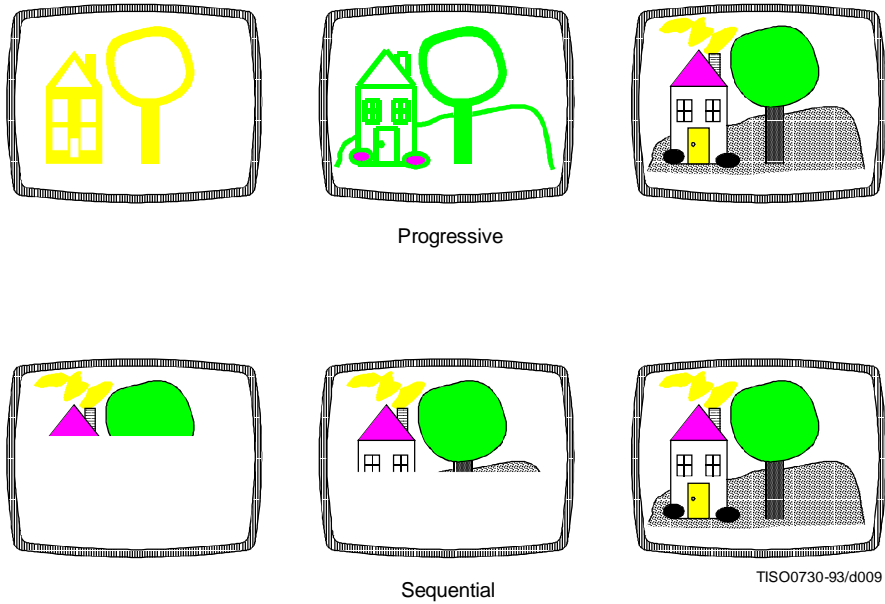


Figure 9 – Progressive versus sequential presentation

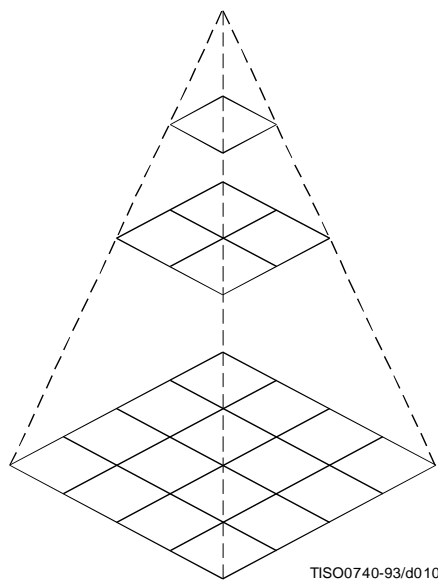


Figure 10 – Hierarchical multi-resolution encoding

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.)

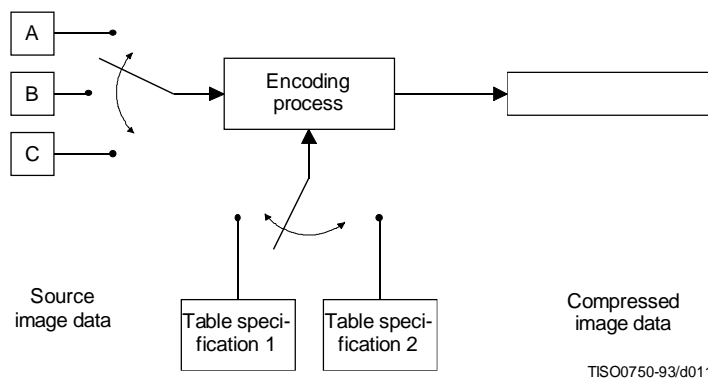


Figure 11 – Component-interleave and table-switching control

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.

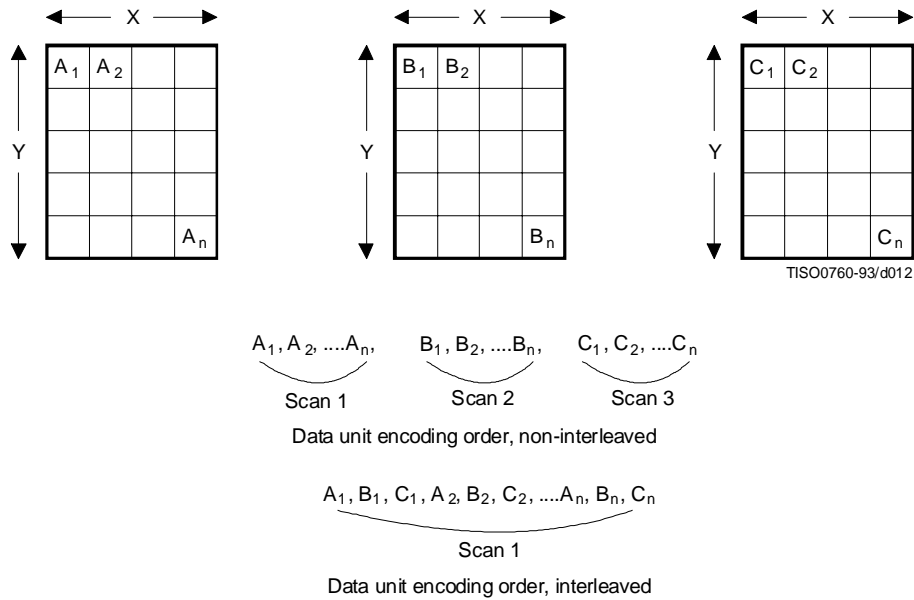


Figure 12 – Interleaved versus non-interleaved encoding order

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.)

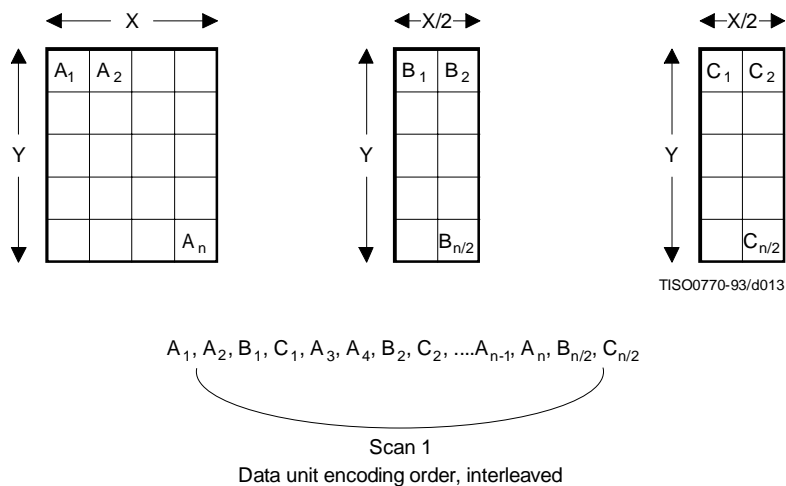


Figure 13 – Interleaved order for components with different dimensions

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)
<ul style="list-style-type: none"> ● 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
<ul style="list-style-type: none"> ● 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
<ul style="list-style-type: none"> ● 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
<ul style="list-style-type: none"> ● 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 N_f 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 $\lceil \cdot \rceil$ 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	$H_0 = 4, V_0 = 1$
Component 1	$H_1 = 2, V_1 = 2$
Component 2	$H_2 = 1, V_2 = 1$

Then $X = 512, Y = 512, H_{max} = 4, V_{max} = 2$, and x_i and y_i for each component are

Component 0	$x_0 = 512, y_0 = 256$
Component 1	$x_1 = 256, y_1 = 512$
Component 2	$x_2 = 128, y_2 = 256$

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.

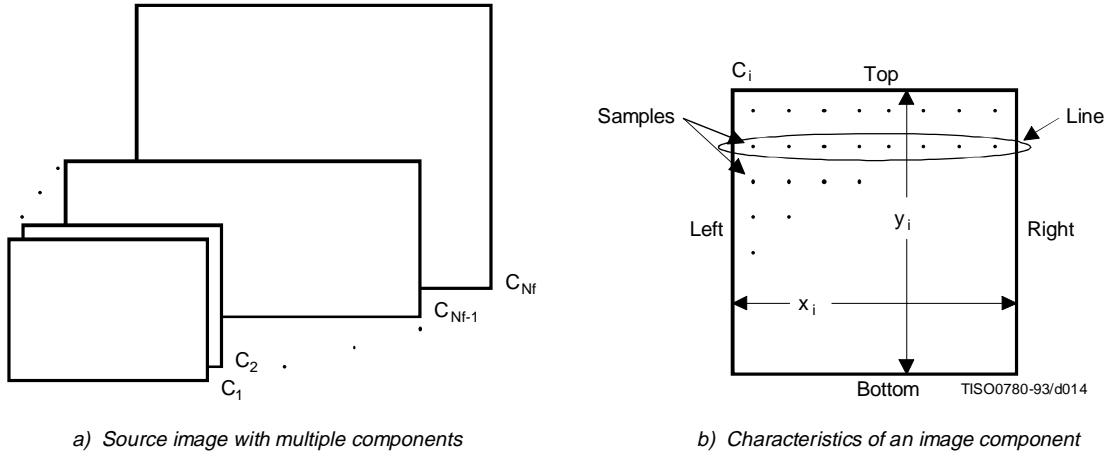


Figure A.1 – Source image characteristics

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 $N_s = 1$, then data from only one source component – the component specified by parameter C_{s1} – shall be present within the scan. This data is non-interleaved by definition. If $N_s > 1$, then data from the N_s components C_{s1} through C_{sN_s} 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 ($N_s = 1$)

When $N_s = 1$ (where N_s 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 $N_s = 1$, regardless of the values of H_1 and V_1 .

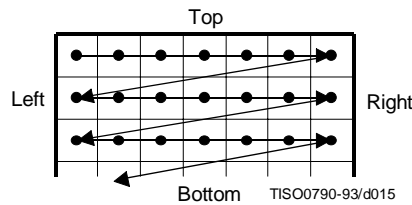


Figure A.2 – Non-interleaved data ordering

A.2.3 Interleaved order (Ns > 1)

When $N_s > 1$, 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, $N_s = 4$, 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.

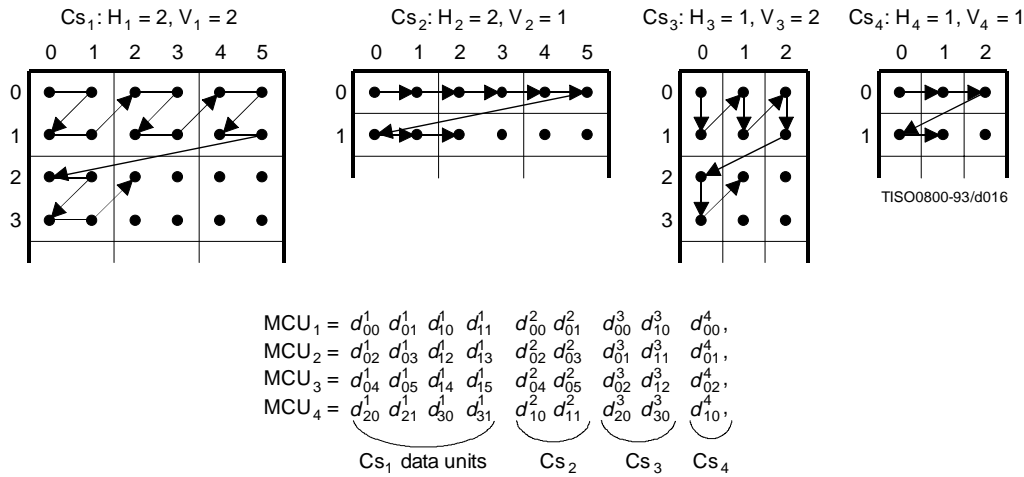


Figure A.3 – Interleaved data ordering example

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.

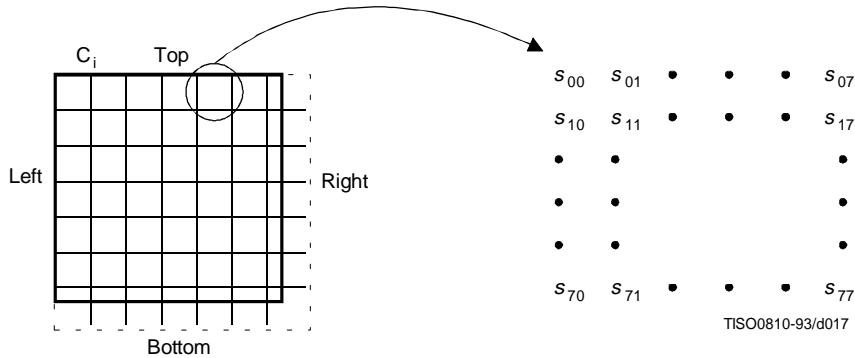


Figure A.4 – Partition and orientation of 8 x 8 sample blocks

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.

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

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

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:

$$Sq_{vu} = \text{round} \left(\frac{S_{vu}}{Q_{vu}} \right)$$

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.

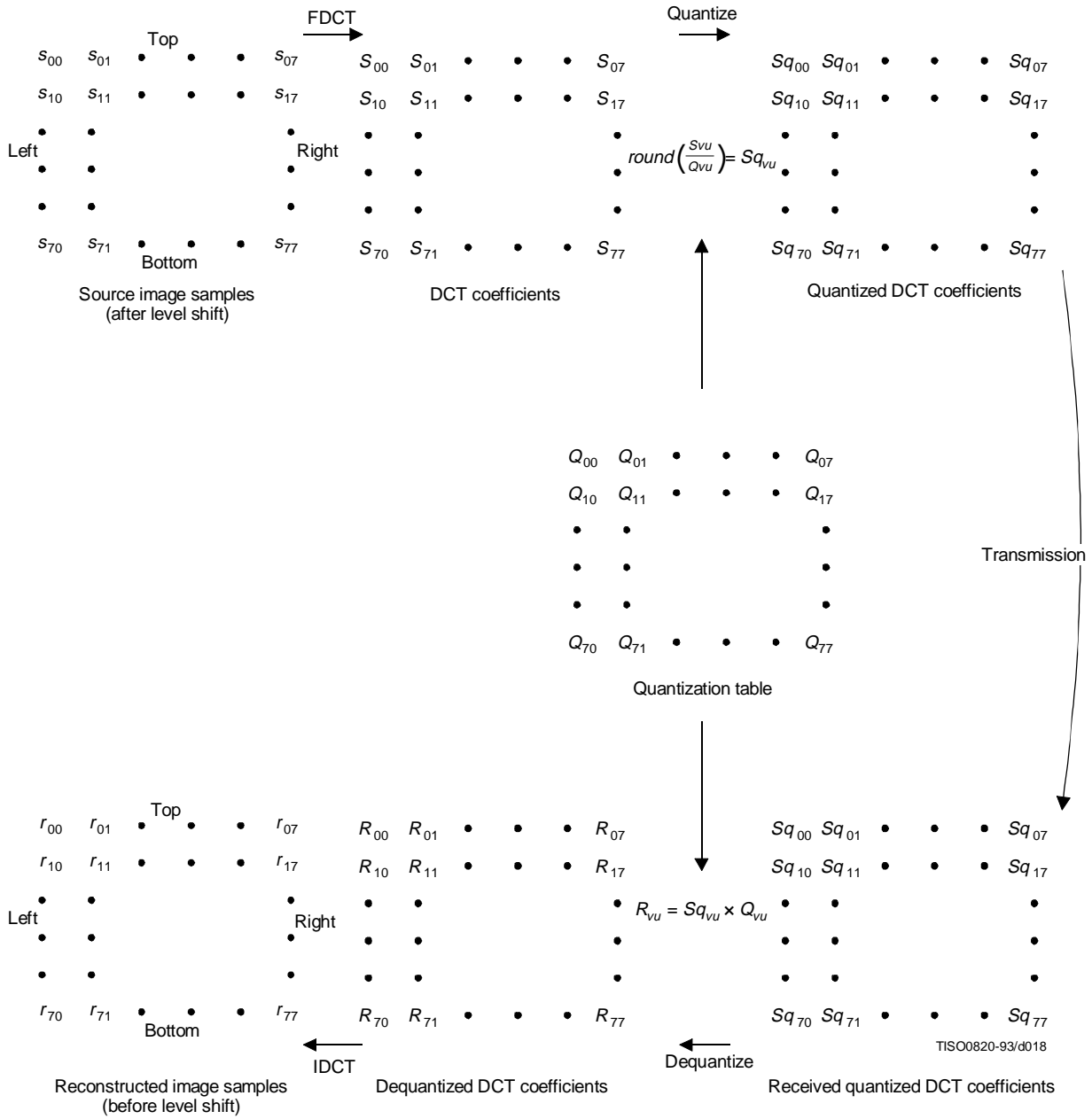


Figure A.5 – Relationship between 8 × 8-block samples and DCT coefficients

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'FFE7' X'FFF0' through X'FFF7' 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:

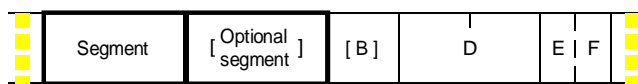
- the required ordering of markers, parameters, and entropy-coded segments;
- identification of optional or conditional constituent parts;
- the name, symbol, and definition of each marker and parameter;
- the allowed values of each parameter;
- any restrictions on the above which are specific to the 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.



TISO0830-93/d019

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.

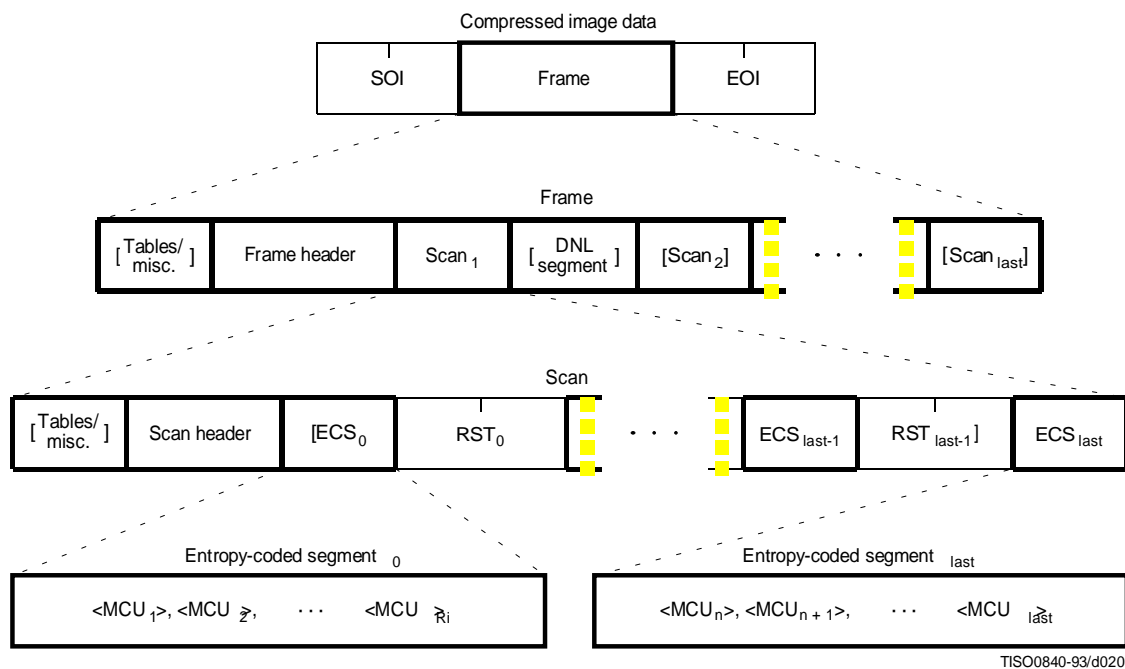


Figure B.2 – Syntax for sequential DCT-based, progressive DCT-based, and lossless modes of operation

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 R_i , each entropy-coded segment except the last one shall contain R_i 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.

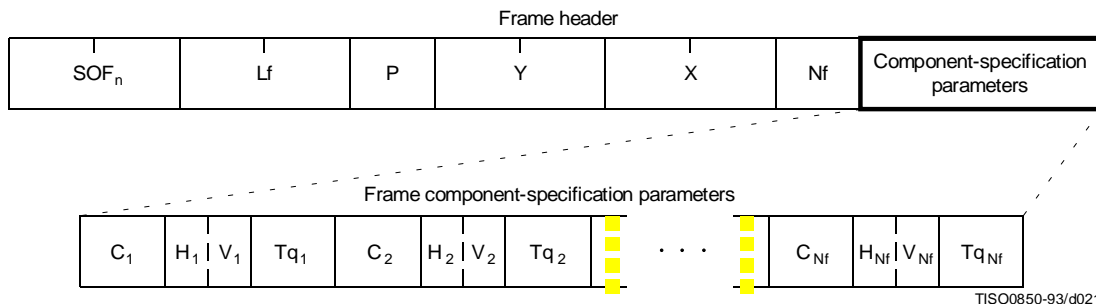


Figure B.3 – Frame header syntax

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.

SOF₀: Baseline DCT

SOF₁: Extended sequential DCT, Huffman coding

SOF₂: Progressive DCT, Huffman coding

SOF₃: Lossless (sequential), Huffman coding

SOF₉: Extended sequential DCT, arithmetic coding

SOF₁₀: Progressive DCT, arithmetic coding

SOF₁₁: 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.

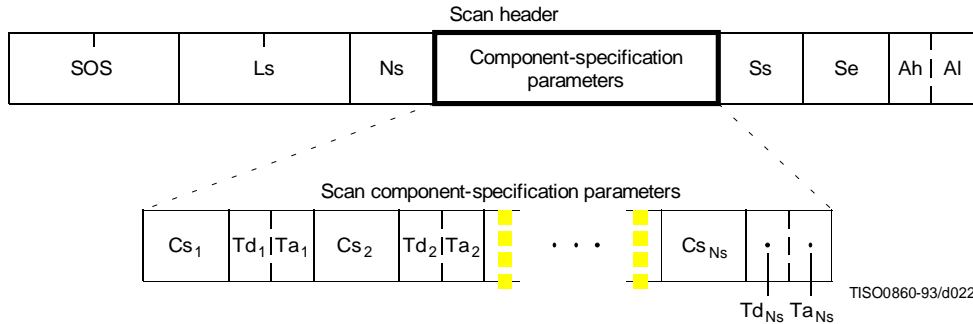


Figure B.4 – Scan header syntax

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 N_f image components specified in the frame parameters shall be the jth 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 Cs₁ first, Cs₂ 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 × V_j is (4 × 1) + (1 × 2) + (2 × 2) = 10.

The value of Cs_j shall be different from the values of Cs₁ to Cs_{j-1}.

T_{dj}: 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 C_{s_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.

T_{aj}: 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 C_{s_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.

S_s: 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.

S_e: 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.

A_h: 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 S_s and S_e. 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.

A_l: 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 S_s and S_e. 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, T_{dj} and T_{aj}, 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		
L _s	16	6 + 2 × N _s			
N _s	8	1-4			
C _{s_j}	8	0-255 ^{a)}			
T _{dj}	4	0-1	0-3	0-3	0-3
T _{aj}	4	0-1	0-3	0-3	0
S _s	8	0	0	0-63	1-7 ^{b)}
S _e	8	63	63	S _s -63 ^{c)}	0
A _h	4	0	0	0-13	0
A _l	4	0	0	0-13	0-15

a) C_{s_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 S_s 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.

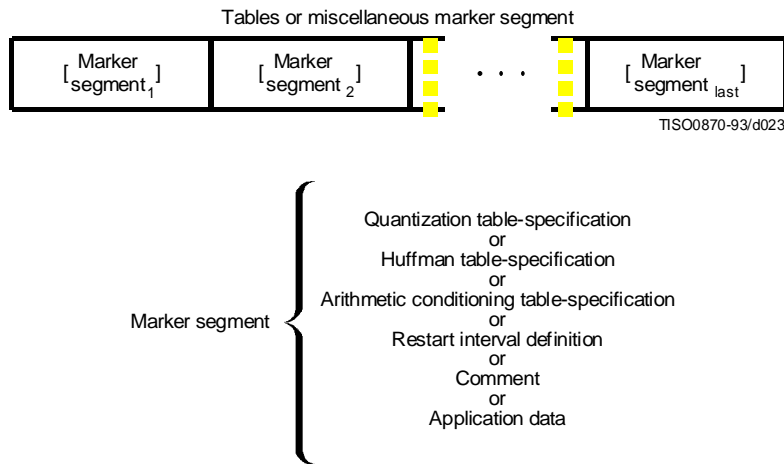


Figure B.5 – Tables/miscellaneous marker segment syntax

B.2.4.1 Quantization table-specification syntax

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

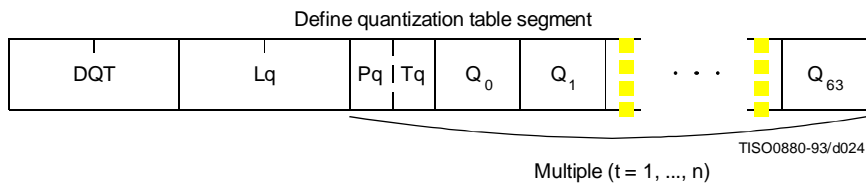


Figure B.6 – Quantization table syntax

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).

P_q: 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. P_q shall be zero for 8 bit sample precision P (see B.2.2).

T_q: 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		
L _q	16	$2 + \sum_{t=1}^n (65 + 64 \times Pq(t))$			Undefined
P _q	4	0	0, 1	0, 1	Undefined
T _q	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.

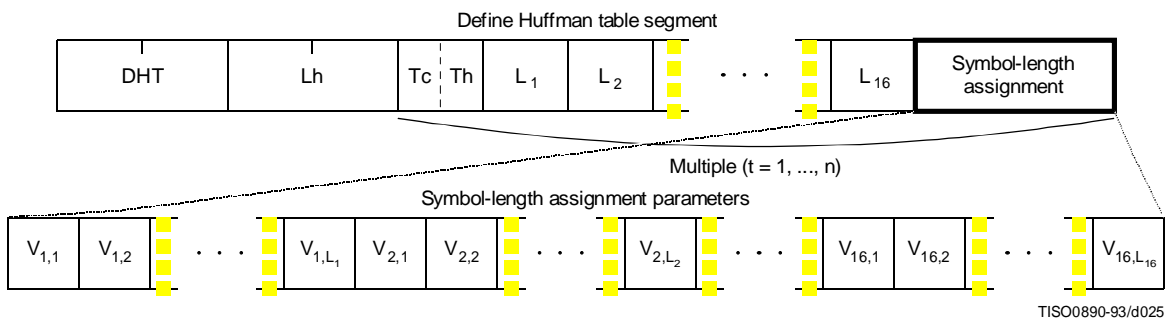


Figure B.7 – Huffman table syntax

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	0
Th	4	0, 1	0-3		
L _i	8	0-255			
V _{i,j}	8	0-255			

The value n in Table B.5 is the number of Huffman tables specified in the DHT marker segment. The value m_t is the number of parameters which follow the 16 L_i(t) parameters for Huffman table t, and is given by:

$$m_t = \sum_{i=1}^{16} L_i$$

In general, m_t is different for each table.

Once a Huffman table has been defined for a particular destination, it replaces the previous tables stored in that destination and shall be used when referenced, in the remaining scans of the current image and in subsequent images represented in the abbreviated format for compressed image data. If a table has never been defined for a particular destination, then when this destination is specified in a scan header, the results are unpredictable.

B.2.4.3 Arithmetic conditioning table-specification syntax

Figure B.8 specifies the marker segment which defines one or more arithmetic coding conditioning table specifications. These replace the default arithmetic coding conditioning tables established by the SOI marker for arithmetic coding processes. (See F.1.4.4.1.4 and F.1.4.4.2.1.)

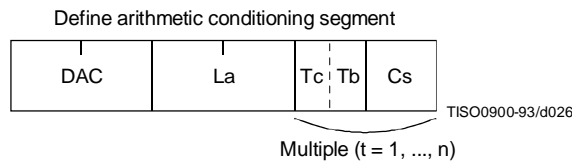


Figure B.8 – Arithmetic conditioning table-specification syntax

The marker and parameters shown in Figure B.8 are defined below. The size and allowed values of each parameter are given in Table B.6.

DAC: Define arithmetic coding conditioning marker – Marks the beginning of the definition of arithmetic coding conditioning parameters.

La: Arithmetic coding conditioning definition length – Specifies the length of all arithmetic coding conditioning parameters shown in Figure B.8 (see B.1.1.4).

Tc: Table class – 0 = DC table or lossless table, 1 = AC table.

Tb: Arithmetic coding conditioning table destination identifier – Specifies one of four possible destinations at the decoder into which the arithmetic coding conditioning table shall be installed.

Cs: Conditioning table value – Value in either the AC or the DC (and lossless) conditioning table. A single value of Cs shall follow each value of Tb. For AC conditioning tables Tc shall be one and Cs shall contain a value of Kx in the range $1 \leq Kx \leq 63$. For DC (and lossless) conditioning tables Tc shall be zero and Cs shall contain two 4-bit parameters, U and L. U and L shall be in the range $0 \leq L \leq U \leq 15$ and the value of Cs shall be $L + 16 \times U$.

The value n in Table B.6 is the number of arithmetic coding conditioning tables specified in the DAC marker segment. The parameters L and U are the lower and upper conditioning bounds used in the arithmetic coding procedures defined for DC coefficient coding and lossless coding. The separate value range 1-63 listed for DCT coding is the Kx conditioning used in AC coefficient coding.

Table B.6 – Arithmetic coding conditioning table-specification parameter sizes and values

Parameter	Size (bits)	Values			
		Sequential DCT		Progressive DCT	Lossless
		Baseline	Extended		
La	16	Undefined	$2 + 2 \times n$		
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.

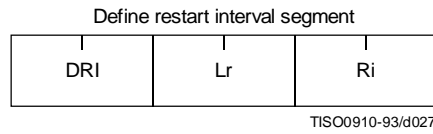


Figure B.9 – Restart interval definition syntax

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			$n \times \text{MCUR}$

B.2.4.5 Comment syntax

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

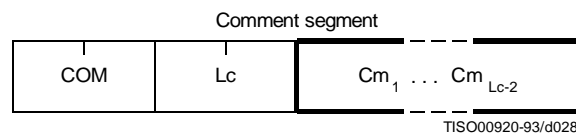


Figure B.10 – Comment segment syntax

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.

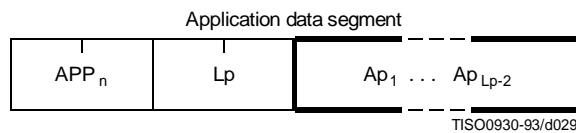


Figure B.11 – Application data syntax

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.

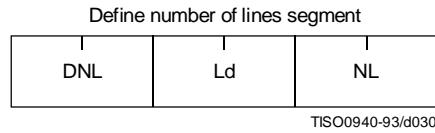


Figure B.12 – Define number of lines syntax

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: SOF₀, SOF₁, SOF₂, SOF₃, SOF₉, SOF₁₀ and SOF₁₁. The differential frames shall use one of the processes specified for SOF₅, SOF₆, SOF₇, SOF₁₃, SOF₁₄ and SOF₁₅. 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.

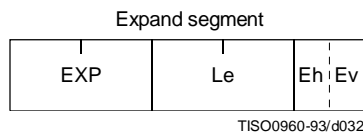


Figure B.14 – Syntax of the expand segment

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).

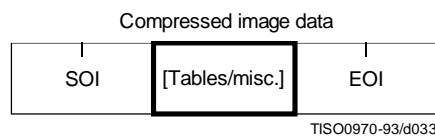


Figure B.15 – Abbreviated format for table-specification data syntax

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.

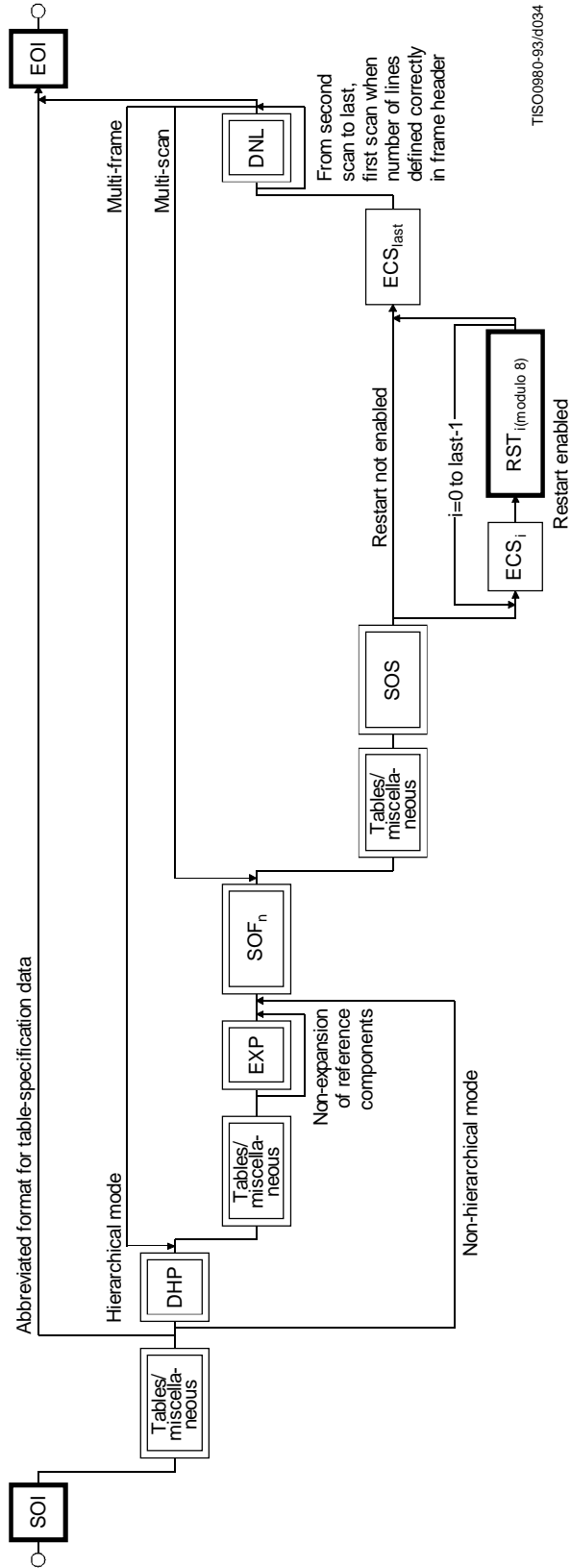
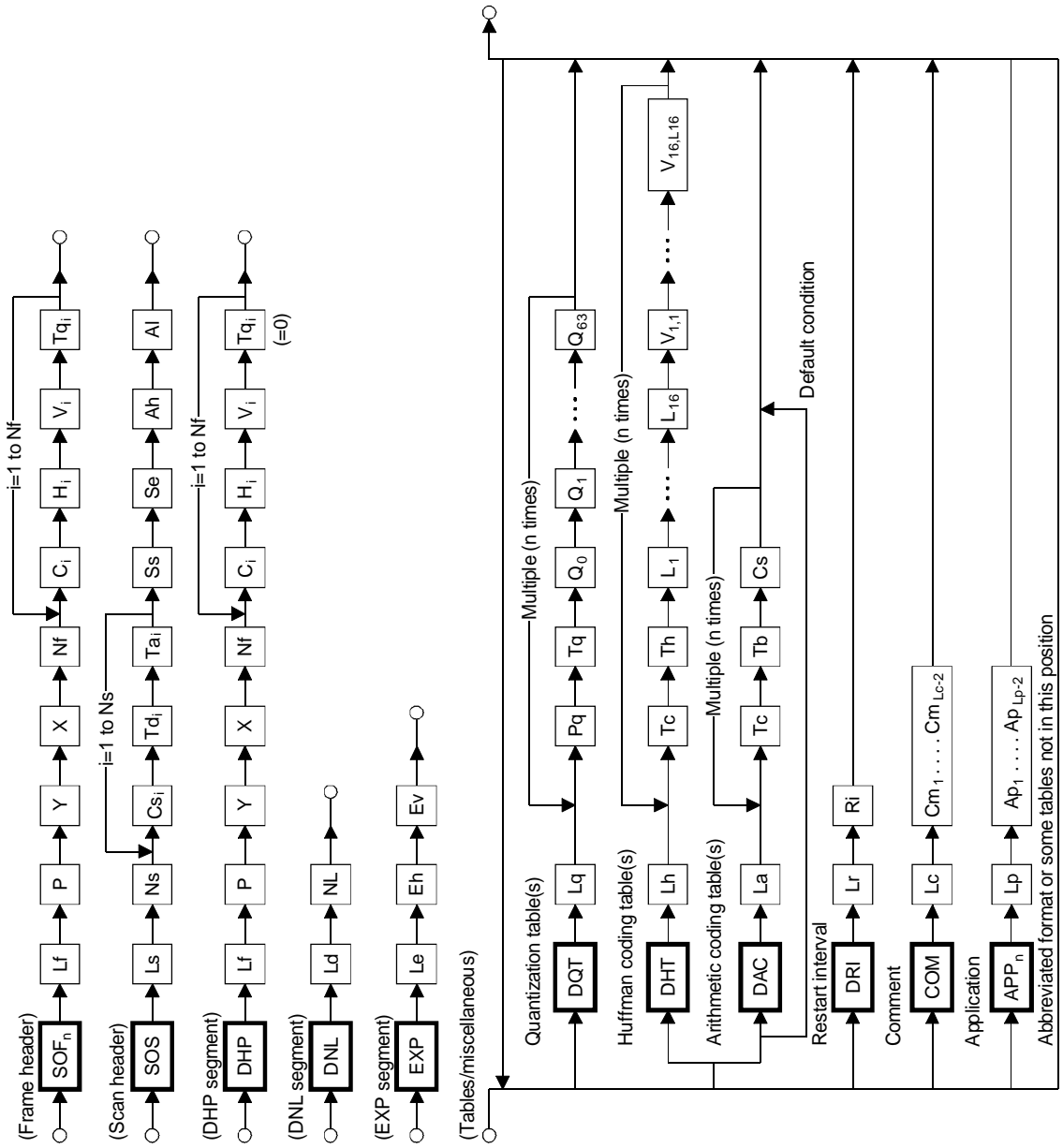


Figure B.16 – Flow of compressed data syntax



T1500990-93/4035

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.

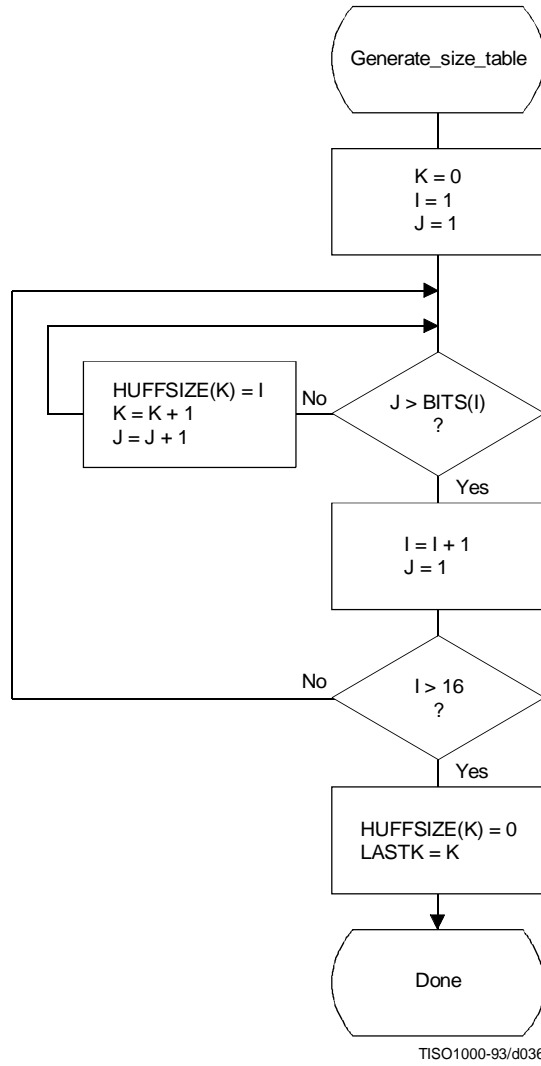


Figure C.1 – Generation of table of Huffman code sizes

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.

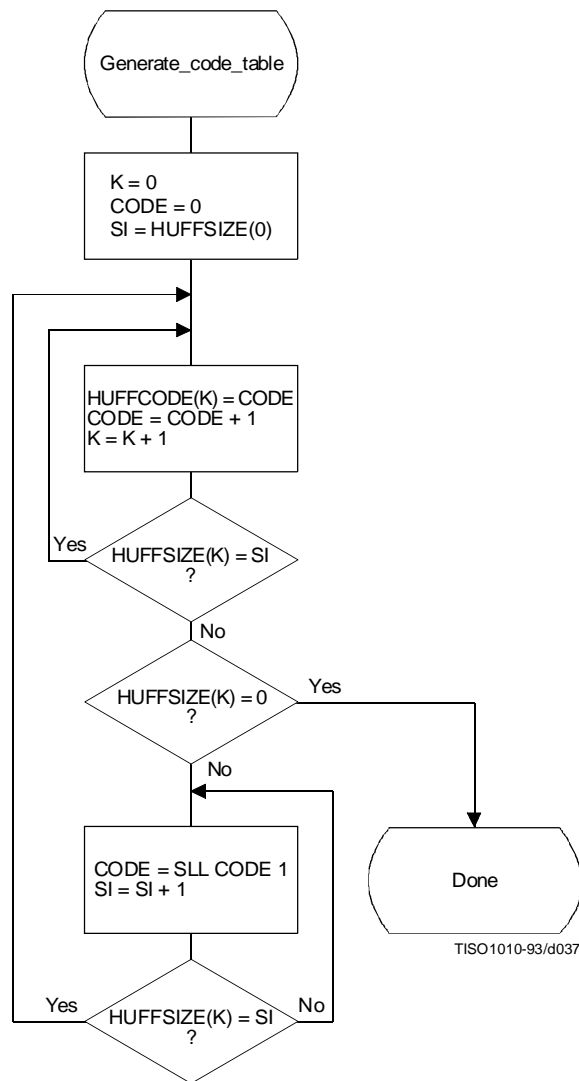


Figure C.2 – Generation of table of Huffman codes

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, EHUFCE and EHUFSE, are created by reordering the codes specified by HUFFCODE and HUFFSIZE according to the symbol values assigned to each code in HUFFVAL.

Figure C.3 illustrates this ordering procedure.

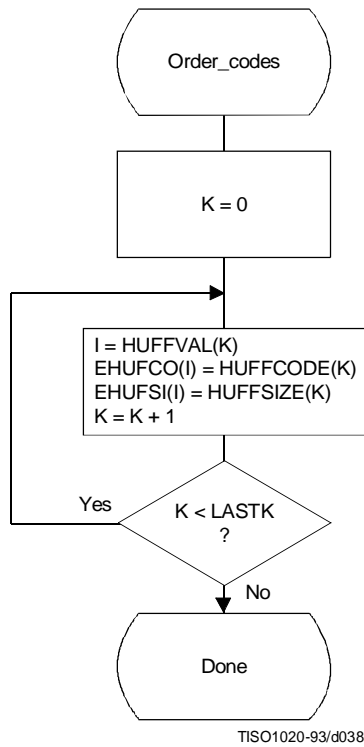


Figure C.3 – Ordering procedure for encoding procedure code tables

C.3 Bit ordering within bytes

The root of a Huffman code is placed toward the MSB (most-significant-bit) of the byte, and successive bits are placed in the direction MSB to LSB (least-significant-bit) of the byte. Remaining bits, if any, go into the next byte following the same rules.

Integers associated with Huffman codes are appended with the MSB adjacent to the LSB of the preceding Huffman code.

Annex D

Arithmetic coding

(This annex forms an integral part of this Recommendation | International Standard)

An adaptive binary arithmetic coding procedure may be used for entropy coding in any of the coding processes except the baseline sequential process. Coding models for adaptive binary arithmetic coding are defined in Annexes F, G, and H. In this annex the arithmetic encoding and decoding procedures used in those models are defined.

In K.4 a simple test example is given which should be helpful in determining if a given implementation is correct.

NOTE – There is **no requirement** in this Specification that any encoder or decoder shall implement the procedures in precisely the manner specified by the flow charts in this annex. It is necessary only that an encoder or decoder implement the **function** specified in this annex. The sole criterion for an encoder or decoder to be considered in compliance with this Specification is that it satisfy the requirements given in clause 6 (for encoders) or clause 7 (for decoders), as determined by the compliance tests specified in Part 2.

D.1 Arithmetic encoding procedures

Four arithmetic encoding procedures are required in a system with arithmetic coding (see Table D.1).

Table D.1 – Procedures for binary arithmetic encoding

Procedure	Purpose
Code_0(S)	Code a “0” binary decision with context-index S
Code_1(S)	Code a “1” binary decision with context-index S
Initenc	Initialize the encoder
Flush	Terminate entropy-coded segment

The “Code_0(S)” and “Code_1(S)” procedures code the 0-decision and 1-decision respectively; S is a context-index which identifies a particular conditional probability estimate used in coding the binary decision. The “Initenc” procedure initializes the arithmetic coding entropy encoder. The “Flush” procedure terminates the entropy-coded segment in preparation for the marker which follows.

D.1.1 Binary arithmetic encoding principles

The arithmetic coder encodes a series of binary symbols, zeros and ones, each symbol representing one possible result of a binary decision.

Each “binary decision” provides a choice between two alternatives. The binary decision might be between positive and negative signs, a magnitude being zero or nonzero, or a particular bit in a sequence of binary digits being zero or one.

The output bit stream (entropy-coded data segment) represents a binary fraction which increases in precision as bytes are appended by the encoding process.

D.1.1.1 Recursive interval subdivision

Recursive probability interval subdivision is the basis for the binary arithmetic encoding procedures. With each binary decision the current probability interval is subdivided into two sub-intervals, and the bit stream is modified (if necessary) so that it points to the base (the lower bound) of the probability sub-interval assigned to the symbol which occurred.

In the partitioning of the current probability interval into two sub-intervals, the sub-interval for the less probable symbol (LPS) and the sub-interval for the more probable symbol (MPS) are ordered such that usually the MPS sub-interval is closer to zero. Therefore, when the LPS is coded, the MPS sub-interval size is added to the bit stream. This coding convention requires that symbols be recognized as either MPS or LPS rather than 0 or 1. Consequently, the size of the LPS sub-interval and the sense of the MPS for each decision must be known in order to encode that decision.

The subdivision of the current probability interval would ideally require a multiplication of the interval by the probability estimate for the LPS. Because this subdivision is done approximately, it is possible for the LPS sub-interval to be larger than the MPS sub-interval. When that happens a “conditional exchange” interchanges the assignment of the sub-intervals such that the MPS is given the larger sub-interval.

Since the encoding procedure involves addition of binary fractions rather than concatenation of integer code words, the more probable binary decisions can sometimes be coded at a cost of much less than one bit per decision.

D.1.1.2 Conditioning of probability estimates

An adaptive binary arithmetic coder requires a statistical model – a model for selecting conditional probability estimates to be used in the coding of each binary decision. When a given binary decision probability estimate is dependent on a particular feature or features (the context) already coded, it is “conditioned” on that feature. The conditioning of probability estimates on previously coded decisions must be identical in encoder and decoder, and therefore can use only information known to both.

Each conditional probability estimate required by the statistical model is kept in a separate storage location or “bin” identified by a unique context-index S . The arithmetic coder is adaptive, which means that the probability estimates at each context-index are developed and maintained by the arithmetic coding system on the basis of prior coding decisions for that context-index.

D.1.2 Encoding conventions and approximations

The encoding procedures use fixed precision integer arithmetic and an integer representation of fractional values in which $X'8000'$ can be regarded as the decimal value 0.75. The probability interval, A , is kept in the integer range $X'8000' \leq A < X'10000'$ by doubling it whenever its integer value falls below $X'8000'$. This is equivalent to keeping A in the decimal range $0.75 \leq A < 1.5$. This doubling procedure is called renormalization.

The code register, C , contains the trailing bits of the bit stream. C is also doubled each time A is doubled. Periodically – to keep C from overflowing – a byte of data is removed from the high order bits of the C -register and placed in the entropy-coded segment.

Carry-over into the entropy-coded segment is limited by delaying $X'FF'$ output bytes until the carry-over is resolved. Zero bytes are stuffed after each $X'FF'$ byte in the entropy-coded segment in order to avoid the accidental generation of markers in the entropy-coded segment.

Keeping A in the range $0.75 \leq A < 1.5$ allows a simple arithmetic approximation to be used in the probability interval subdivision. Normally, if the current estimate of the LPS probability for context-index S is $Qe(S)$, precise calculation of the sub-intervals would require:

$$\begin{array}{ll} Qe(S) \times A & \text{Probability sub-interval for the LPS;} \\ A - (Qe(S) \times A) & \text{Probability sub-interval for the MPS.} \end{array}$$

Because the decimal value of A is of order unity, these can be approximated by

$$\begin{array}{ll} Qe(S) & \text{Probability sub-interval for the LPS;} \\ A - Qe(S) & \text{Probability sub-interval for the MPS.} \end{array}$$

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,	bbbbssss,	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.

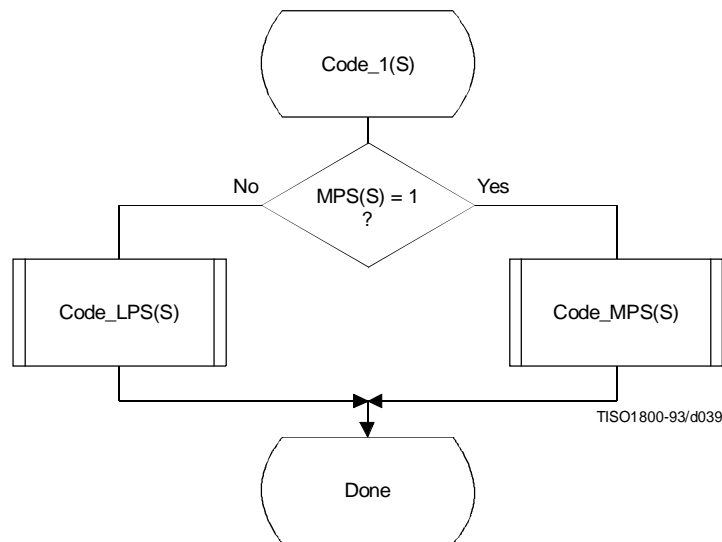


Figure D.1 – Code_1(S) procedure

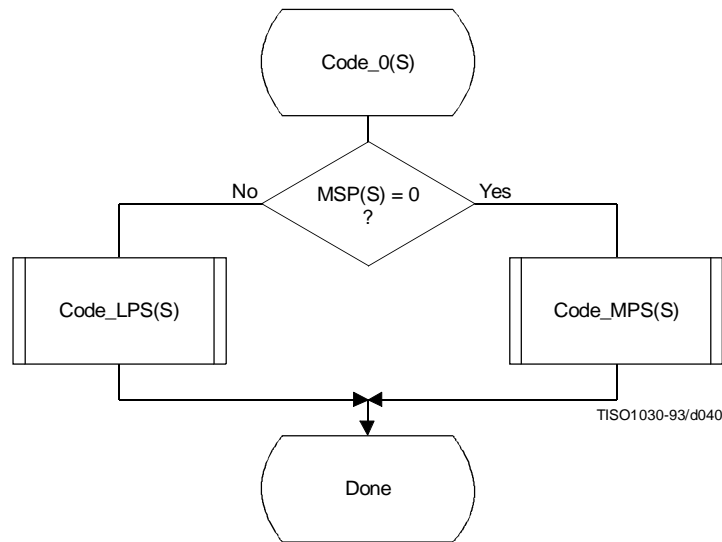


Figure D.2 – Code_0(S) procedure

The context-index S selects a storage location which contains $\text{Index}(S)$, 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, $\text{MPS}(S)$.

For simplicity, the flow charts in this subclause assume that the context storage for each context-index S has an additional storage field for $Q_e(S)$ containing the value of $Q_e(\text{Index}(S))$. If only the value of $\text{Index}(S)$ and $\text{MPS}(S)$ are stored, all references to $Q_e(S)$ should be replaced by $Q_e(\text{Index}(S))$.

The $\text{Code_LPS}(S)$ procedure normally consists of the addition of the MPS sub-interval $A - Q_e(S)$ to the bit stream and a scaling of the interval to the sub-interval, $Q_e(S)$. It is always followed by the procedures for obtaining a new LPS probability estimate ($\text{Estimate_}Q_e(S)_\text{after_LPS}$) 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 $\text{Code_MPS}(S)$ 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 ($\text{Estimate_}Q_e(S)_\text{after_MPS}$) and renormalization (Renorm_e) are required after the coding of the symbol (see Figure D.4).

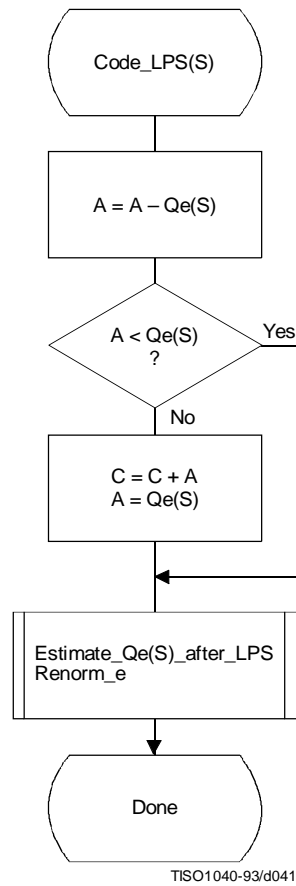
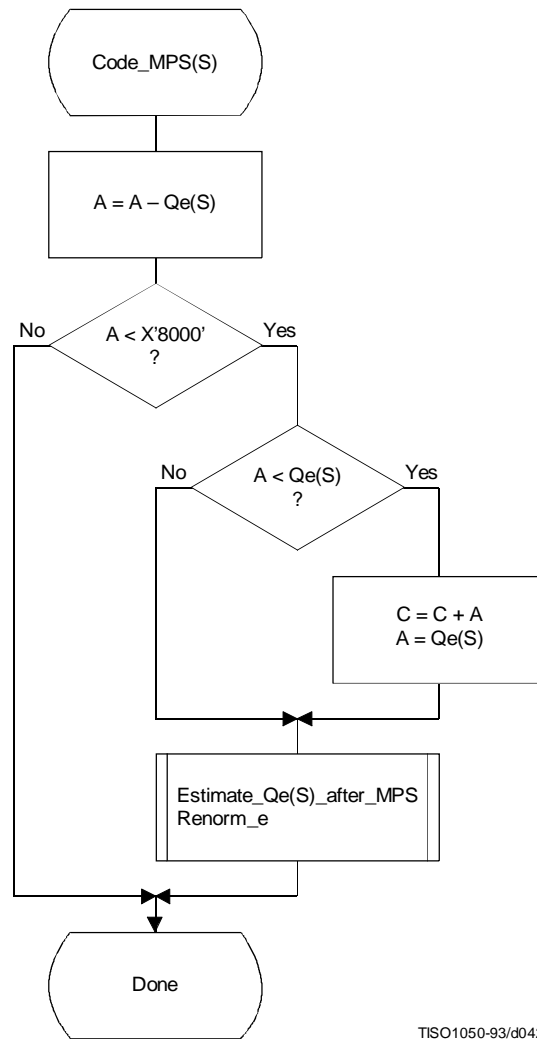


Figure D.3 – Code_LPS(S) procedure with conditional MPS/LPS exchange



TISO1050-93/d042

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 Q_e 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 Q_e index of zero in Table D.3.

The Q_e values listed in Table D.3 are expressed as hexadecimal integers. To approximately convert the 15-bit integer representation of Q_e to a decimal probability, divide the Q_e values by $(4/3) \times (X'8000)$.

Table D.3 – Qe values and probability estimation state machine

Index	Qe _Value	Next_Index		Switch _MPS	Index	Qe _Value	Next_Index		Switch _MPS
		_LPS	_MPS				_LPS	_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.

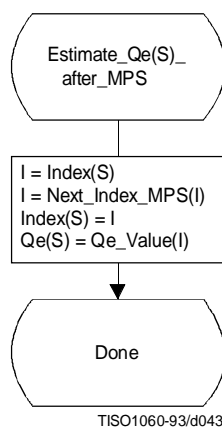


Figure D.5 – Probability estimation on MPS renormalization path

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.

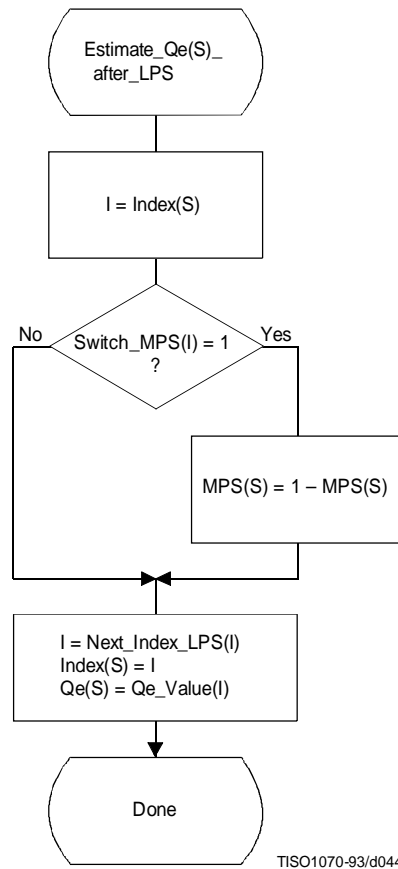


Figure D.6 – Probability estimation on LPS renormalization path

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'.

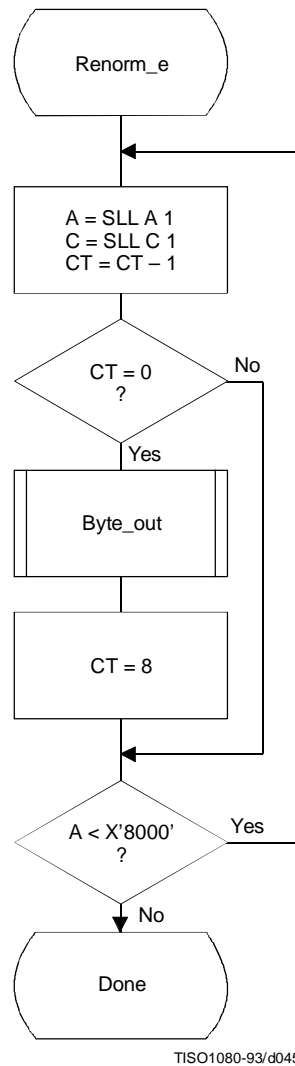
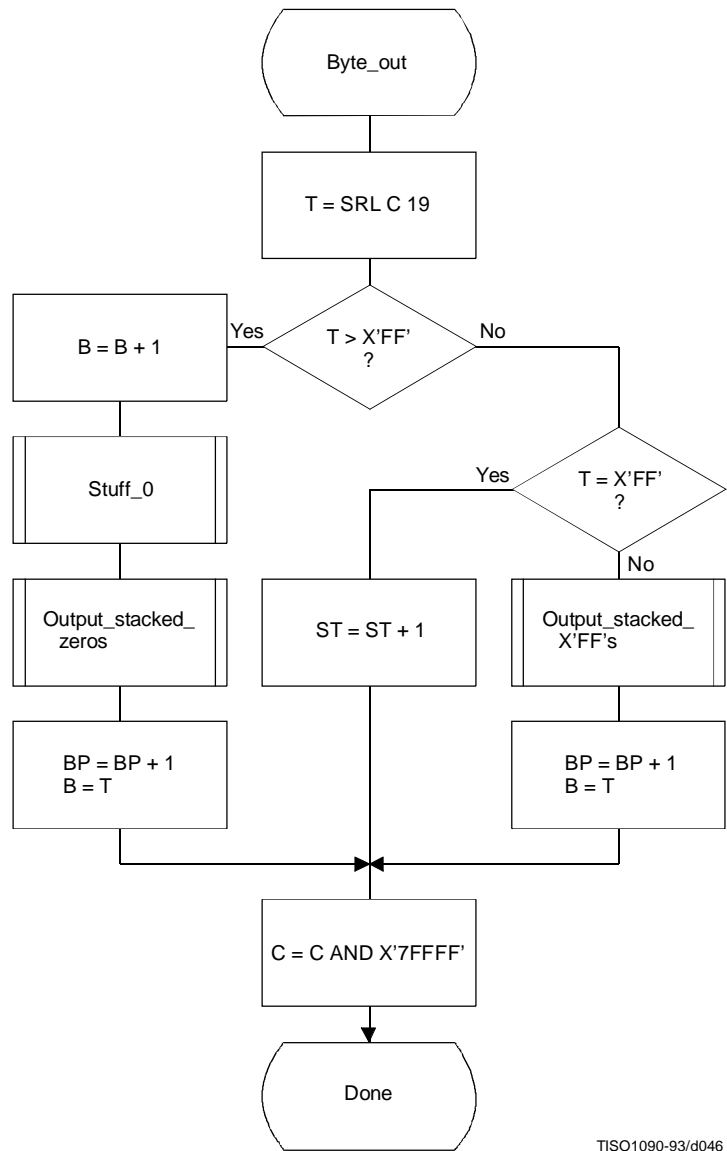


Figure D.7 – Encoder renormalization procedure

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.



TISO1090-93/d046

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.

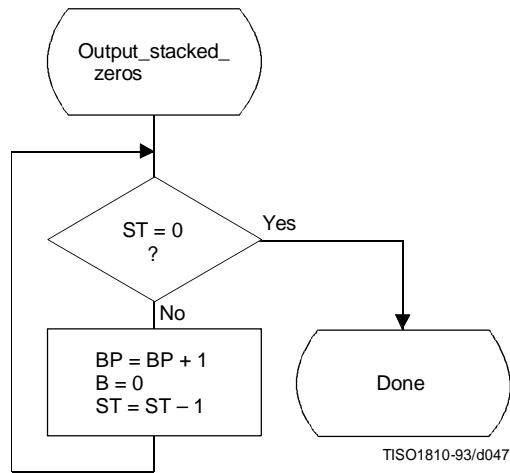


Figure D.9 – Output_stacked_zeros procedure for encoder

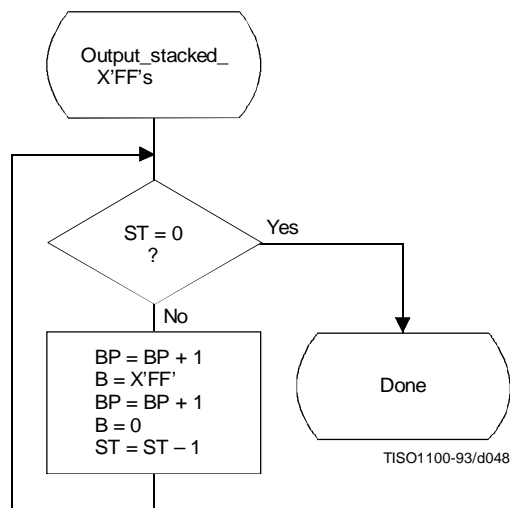


Figure D.10 – Output_stacked_X'FF's procedure for encoder

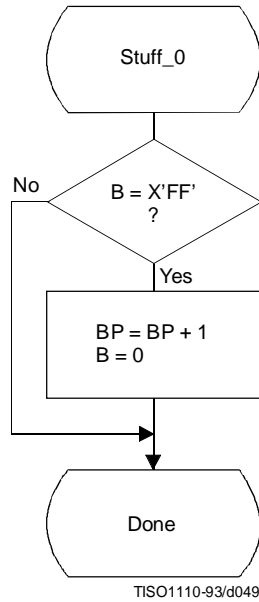


Figure D.11 – Stuff_0 procedure for encoder

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.

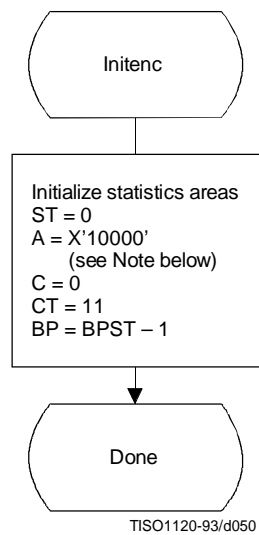


Figure D.12 – Initialization of the encoder

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.

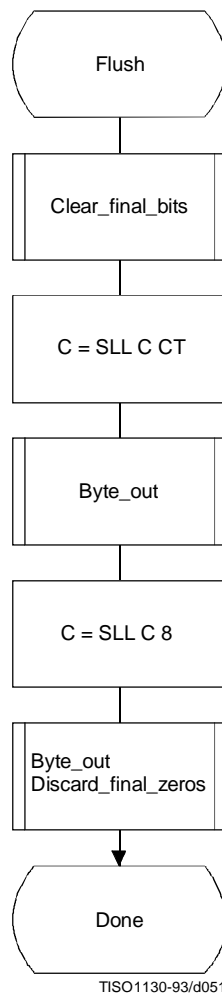


Figure D.13 – Flush procedure

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.

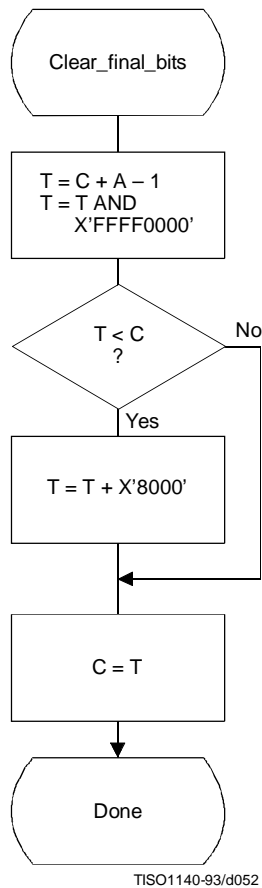


Figure D.14 – Clear_final_bits procedure in Flush

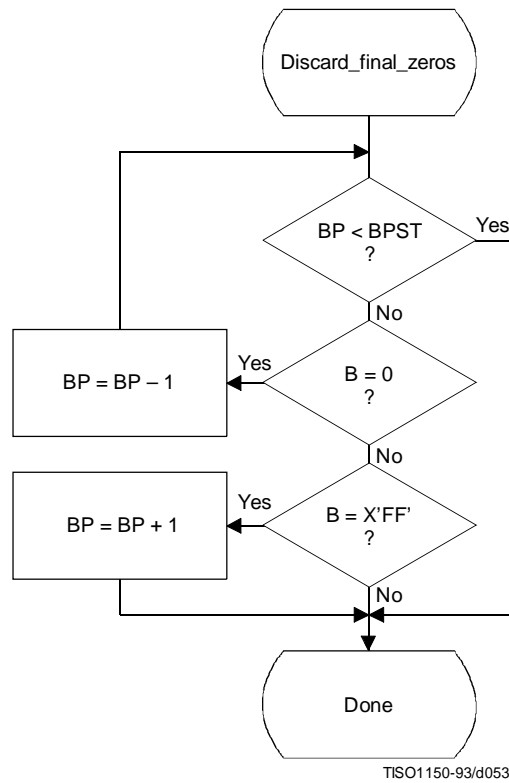


Figure D.15 – Discard_final_zeros procedure in Flush

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	xxxxxxx,	xxxxxxx
C-low	bbbbbbb,	0000000
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 $1 - \text{MPS}(S)$.

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).

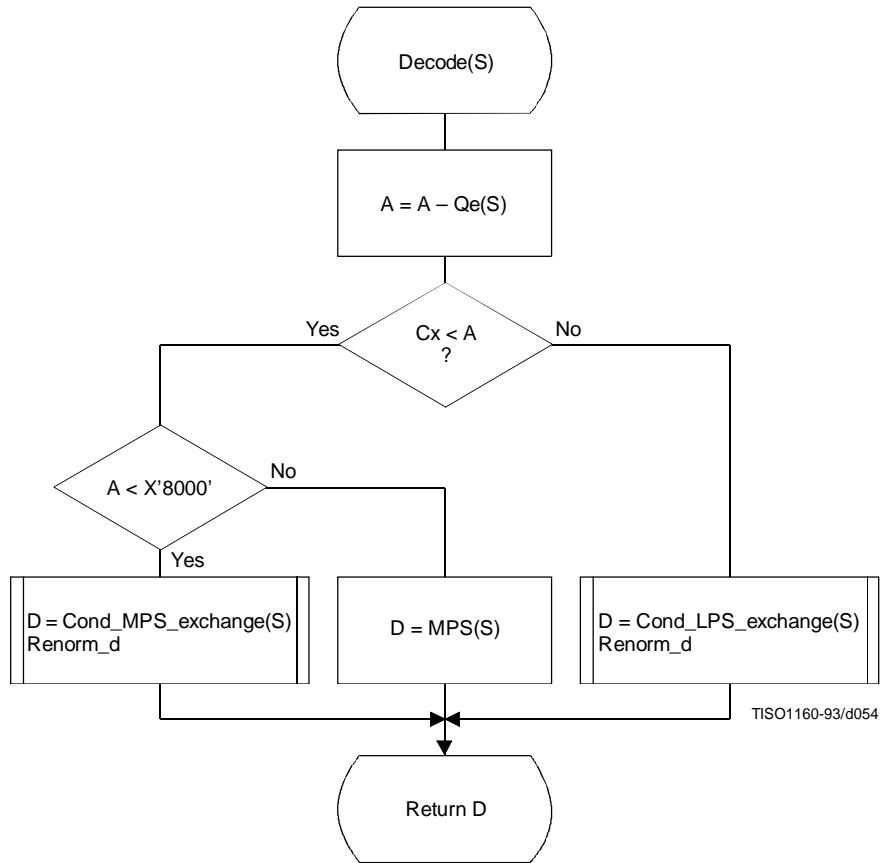


Figure D.16 – Decode(S) procedure

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

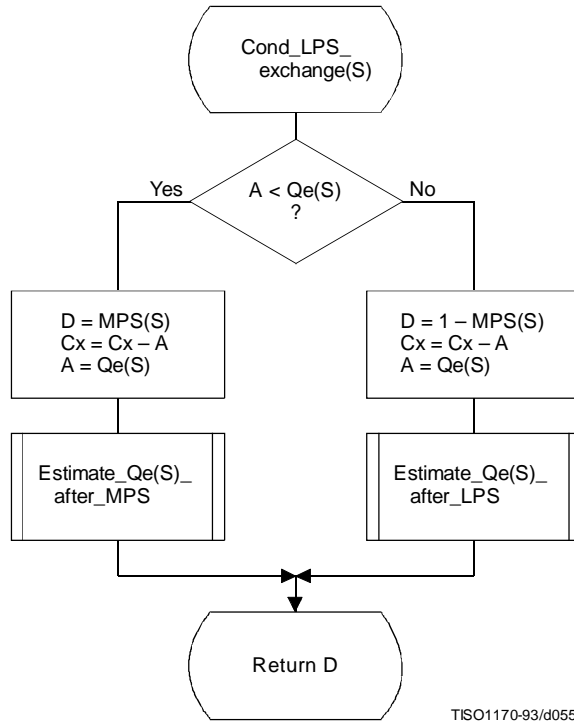


Figure D.17 – Decoder LPS path conditional exchange procedure

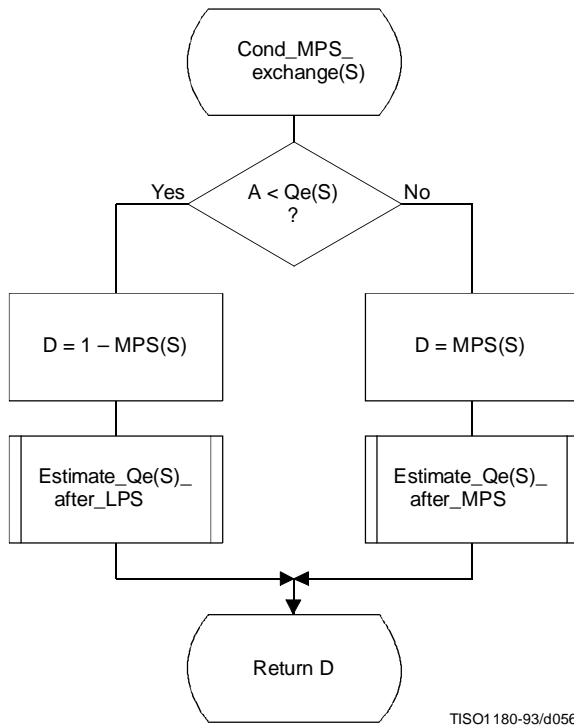


Figure D.18 – Decoder MPS path conditional exchange procedure

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'.

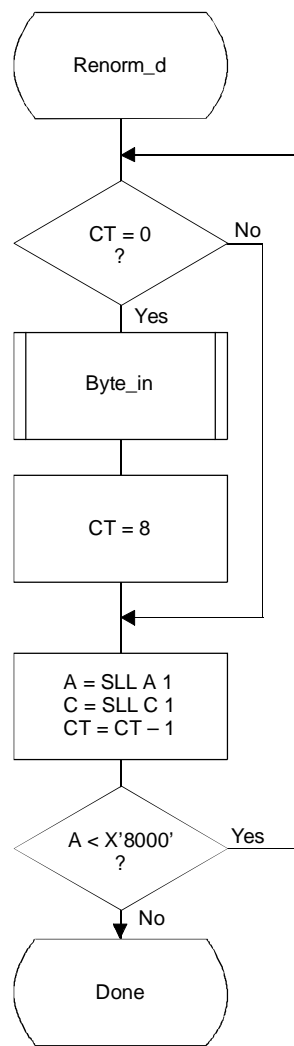


Figure D.19 – Decoder renormalization procedure

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.

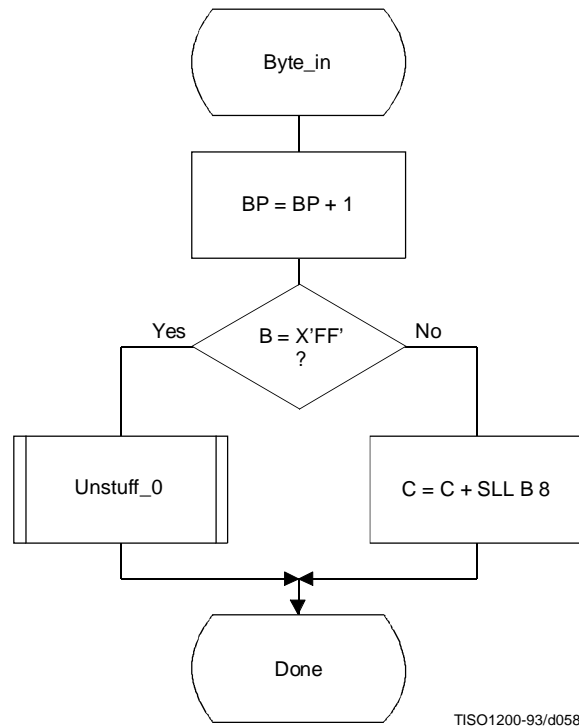


Figure D.20 – Byte_in procedure for decoder

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.

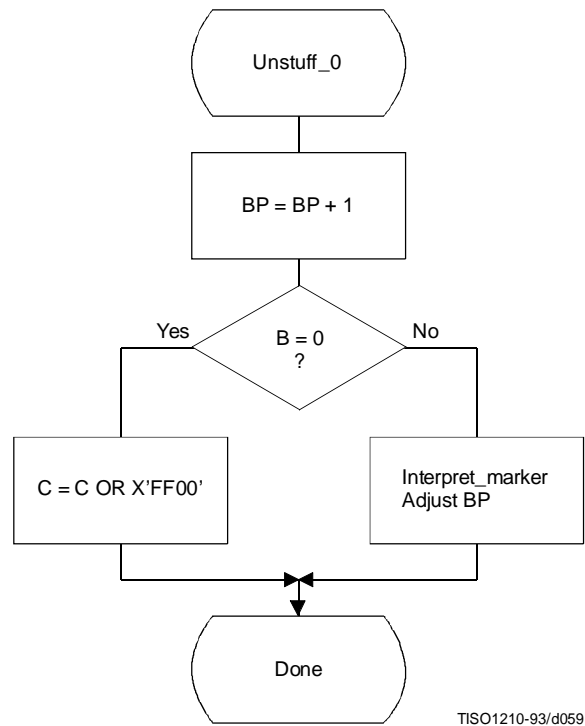


Figure D.21 – Unstuff_0 procedure for decoder

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.

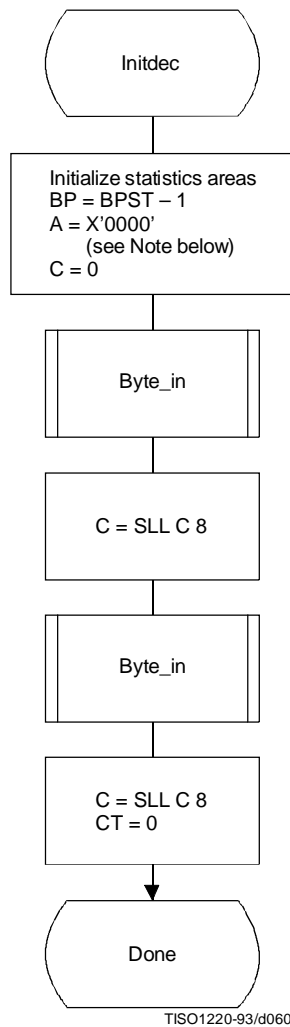


Figure D.22 – Initialization of the decoder

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.

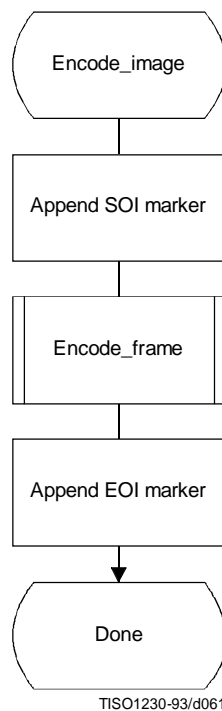


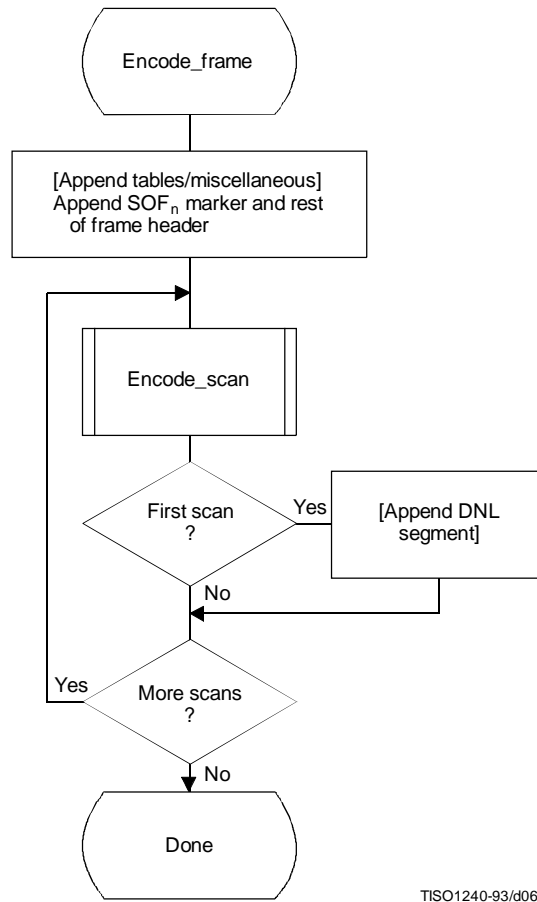
Figure E.1 – Control procedure for encoding an image

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.



TISO1240-93/d062

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.

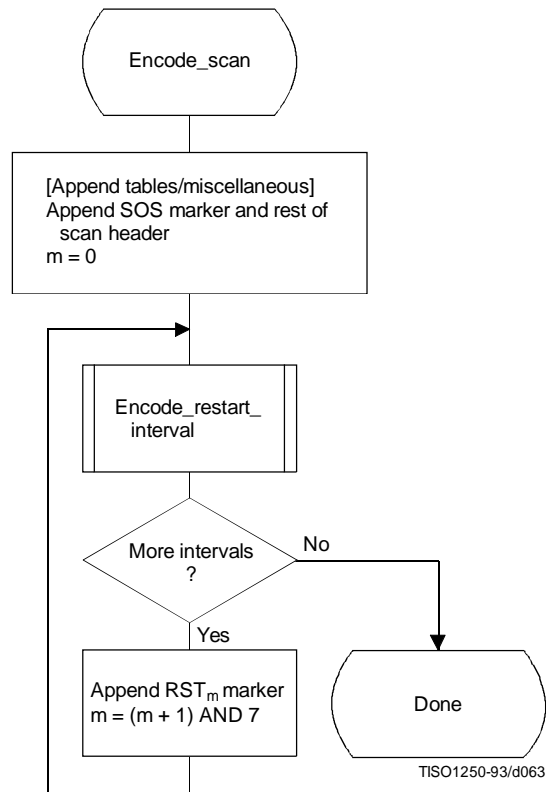


Figure E.3 – Control procedure for encoding a scan

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.

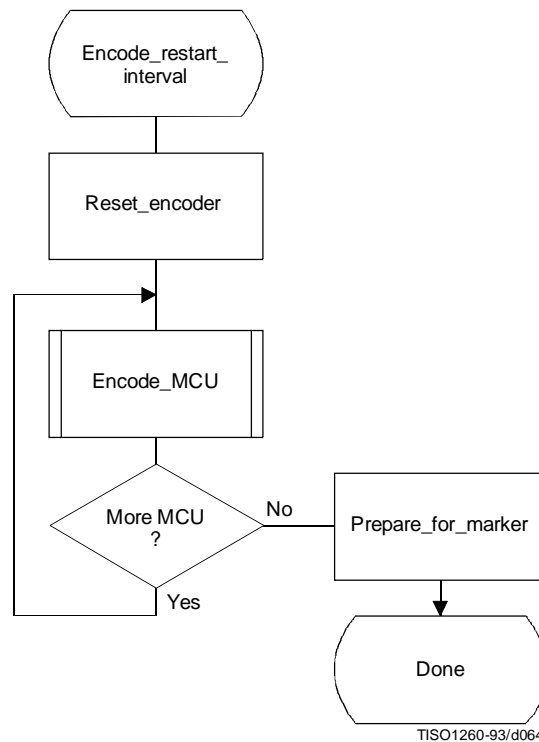


Figure E.4 – Control procedure for encoding a restart interval

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.

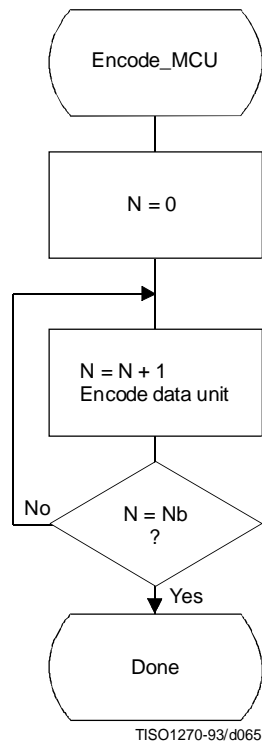


Figure E.5 – Control procedure for encoding a minimum coded unit (MCU)

In Figure E.5, N_b 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.

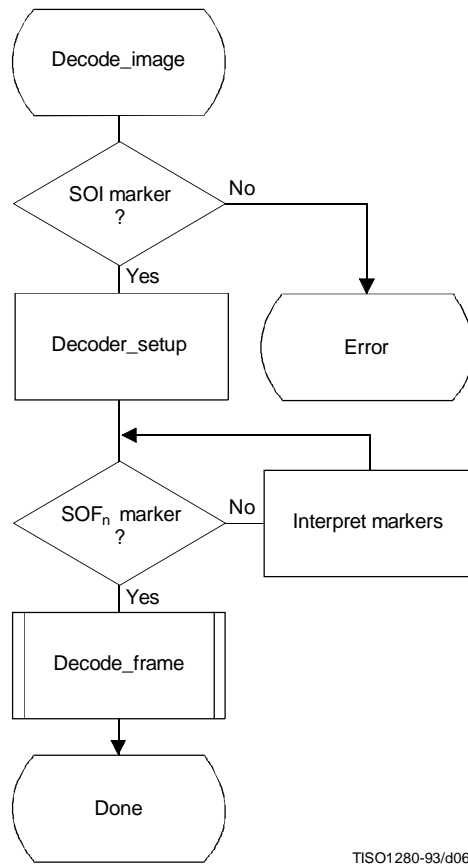
The procedures for encoding a data unit are specified in Annexes F, G, and H.

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 ($R_i = 0$) 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.



TISO1280-93/d066

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

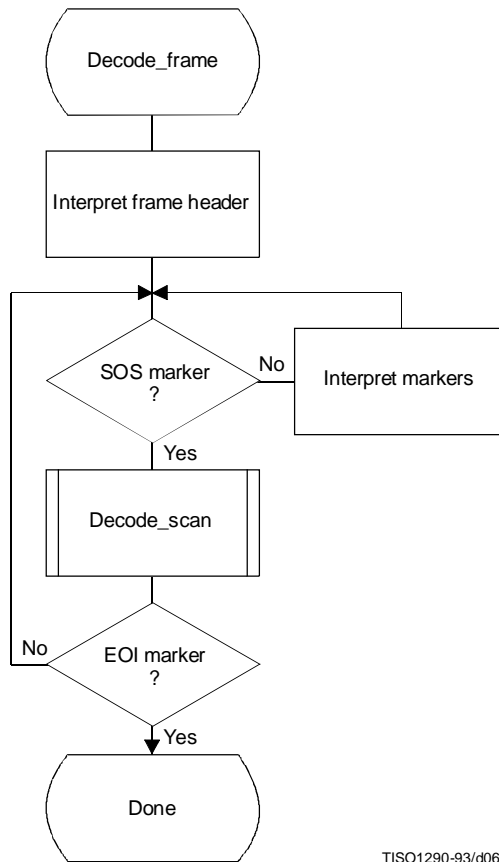
Note that optional X'FF' fill bytes which may precede any marker shall be discarded before determining which marker is present.

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.



TISO1290-93/d067

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.

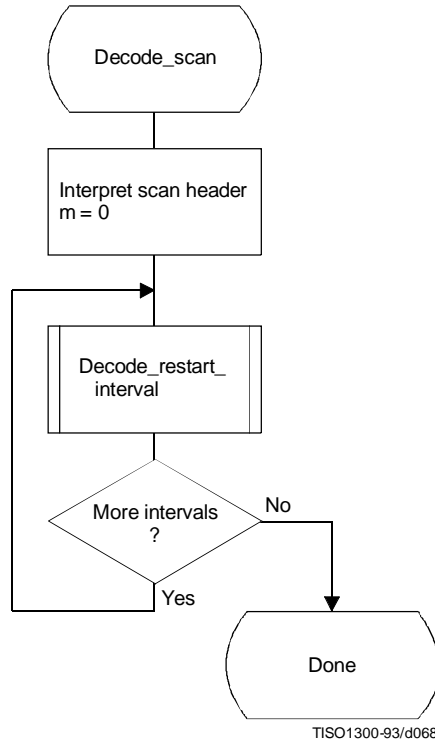


Figure E.8 – Control procedure for decoding a scan

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:

- if arithmetic coding is used, initialize the arithmetic decoder using the “Initdec” procedure described in D.2.7;
- for DCT-based processes, set the DC prediction (PRED) to zero for all components in the scan (see F.2.1.3.1);
- for lossless process, reset the prediction to a default value for all components in the scan (see H.2.1);
- do all other implementation-dependent setups that may be necessary.

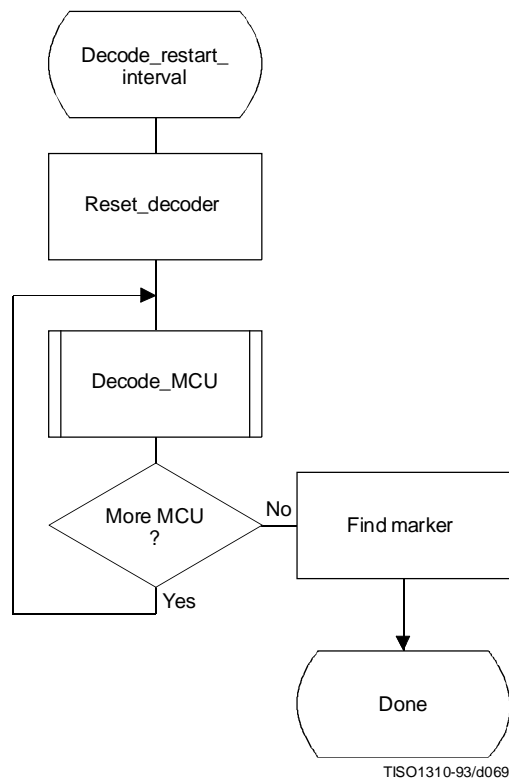


Figure E.9 – Control procedure for decoding a restart interval

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.

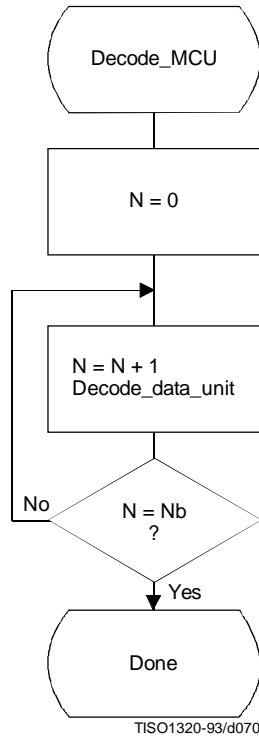


Figure E.10 – Control procedure for decoding a minimum coded unit (MCU)

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:

$$ZZ(0) = Sq_{00}, ZZ(1) = Sq_{01}, ZZ(2) = Sq_{10}, \dots, ZZ(63) = Sq_{77}$$

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, XHUFACO and XHUFASI, 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.

XHUFACO and XHUFASI are generated from the encoder tables EHUFACO and EHUFASI (see Annex C) by appending to the Huffman codes for each difference category the additional bits that completely define the difference. By definition, XHUFACO and XHUFASI have entries for each possible difference value. XHUFACO contains the concatenated bit pattern of the Huffman code and the additional bits field; XHUFASI 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:

$$\text{SIZE} = \text{XHUFASI}(\text{DIFF})$$

$$\text{CODE} = \text{XHUFACO}(\text{DIFF})$$

$$\text{code SIZE bits of CODE}$$

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 XHUFACO and SIZE (length of the code including additional bits) is obtained from XHUFASI, 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

$$\text{RS} = \text{binary 'RRRRSSSS'}$$

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.

		SSSS									
		0	1	2	.	.	.	9	10		
RRRR	0	EOB	COMPOSITE VALUES								
	.	N/A									
	.	N/A									
	15	ZRL									

TISO1330-93/d071

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 Kth 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, EHUF0 (containing the code bits) and EHUF1 (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 EHUF1(X'F0') bits of EHUF0(X'F0')” codes a run of 16 zero coefficients (ZRL code of Figure F.1). The procedure “Code EHUF1(0) bits of EHUF0(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

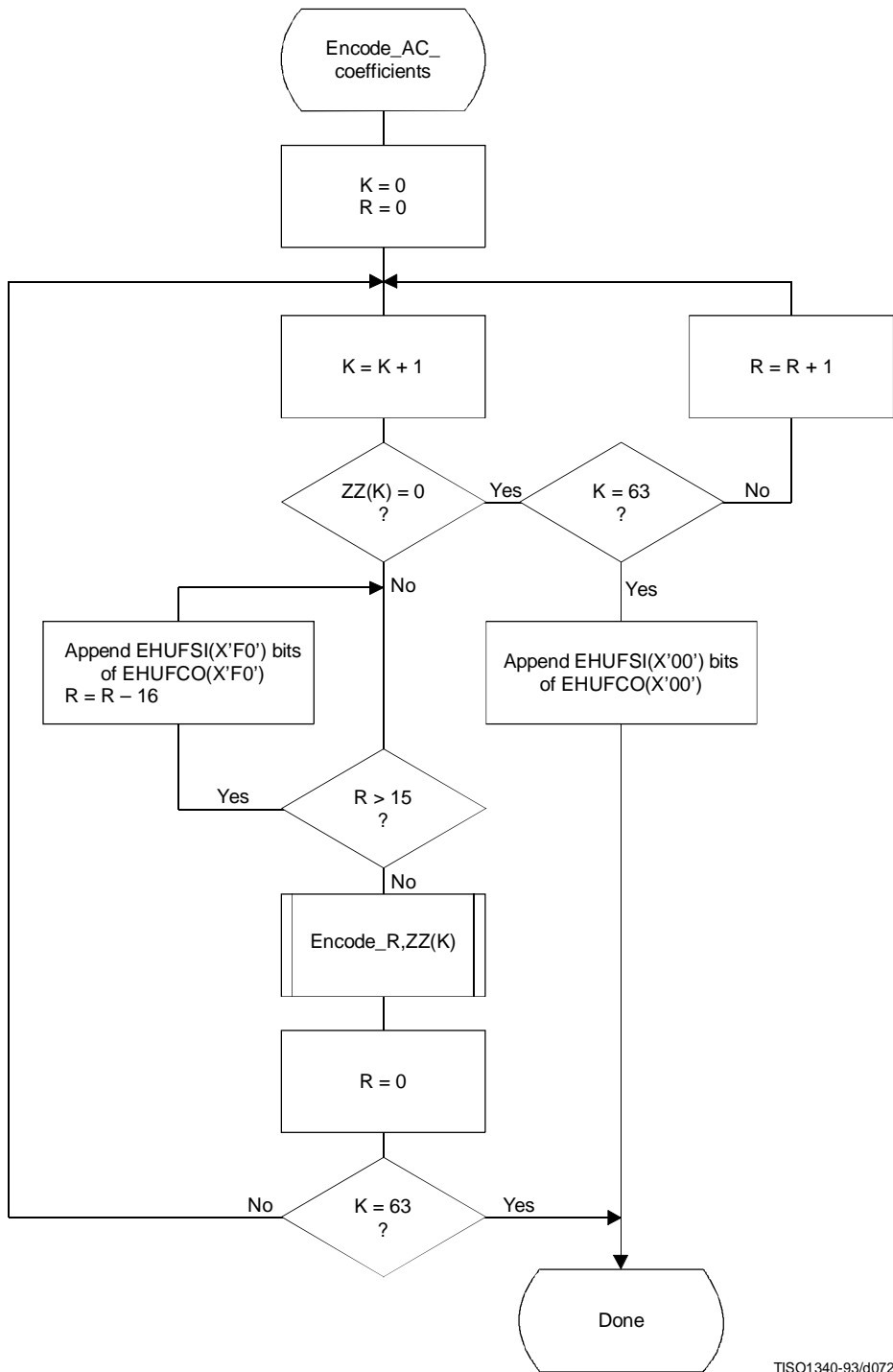
This subclause describes the use of arithmetic coding procedures in the sequential DCT-based encoding process.

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.



TISO1340-93/d072

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

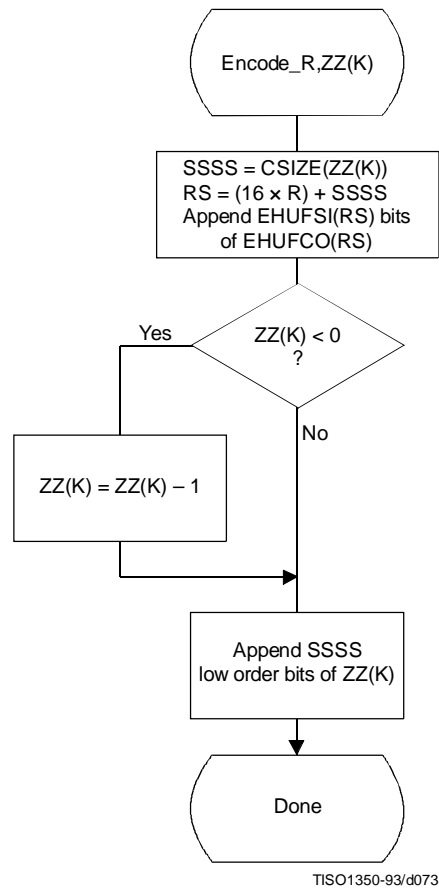


Figure F.3 – Sequential encoding of a non-zero AC coefficient

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 Initec 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 S_0 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(S_0), 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 $ZZ(1, \dots, 63)$. 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.

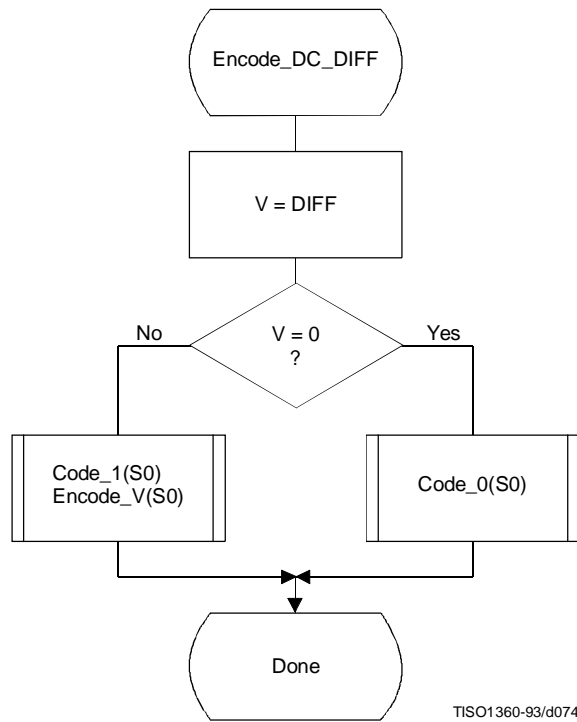


Figure F.4 – Coding model for arithmetic coding of DC difference

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.

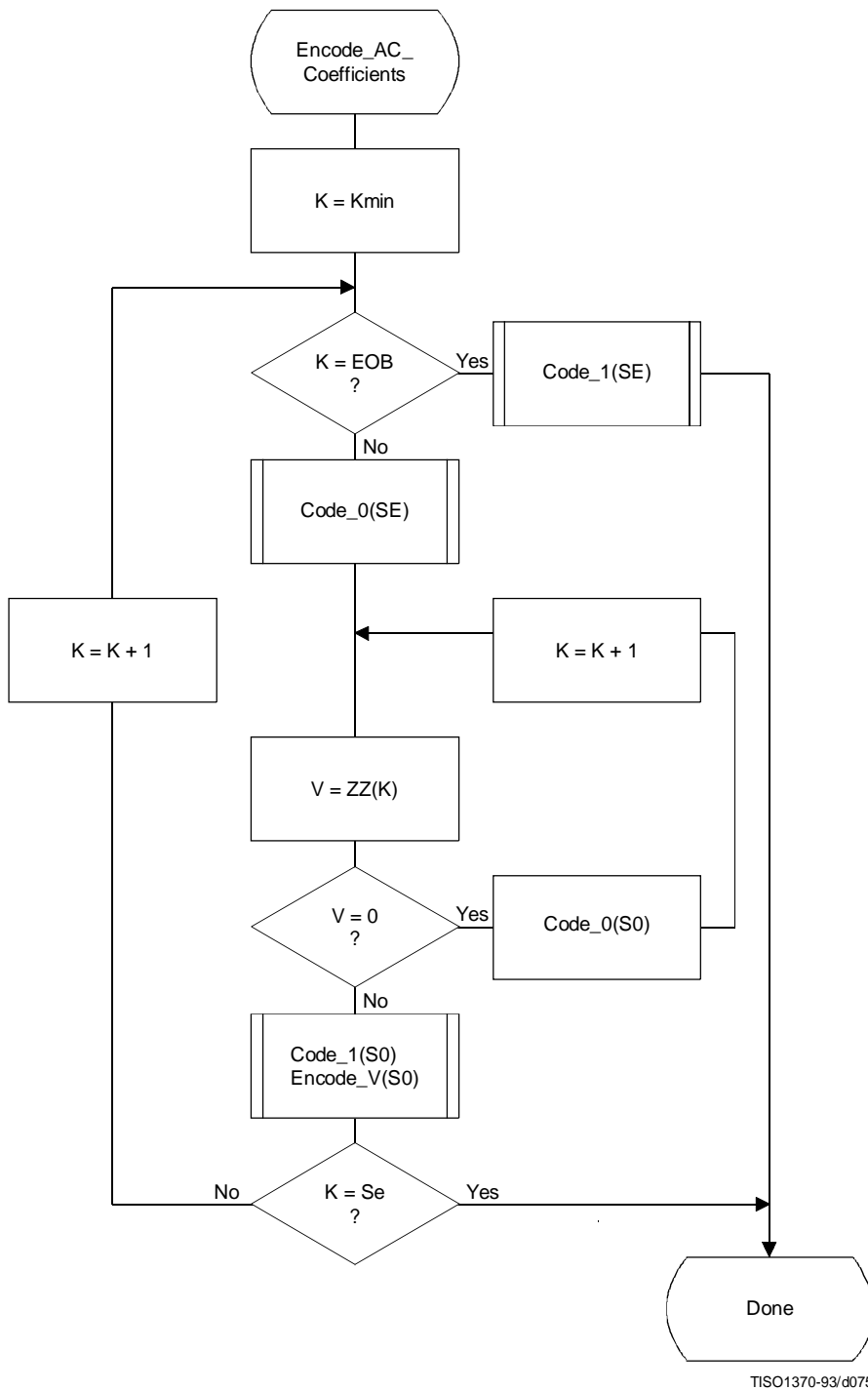


Figure F.5 – AC coding model for arithmetic coding

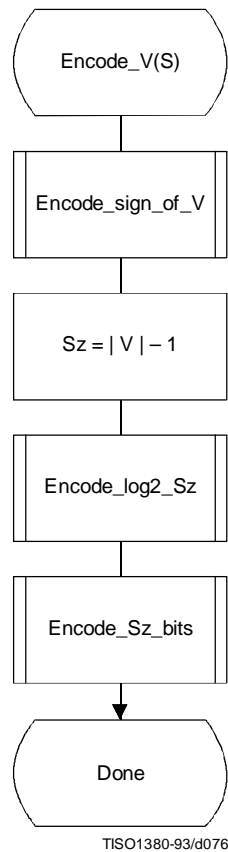


Figure F.6 – Sequence of procedures in encoding non-zero values of V

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.

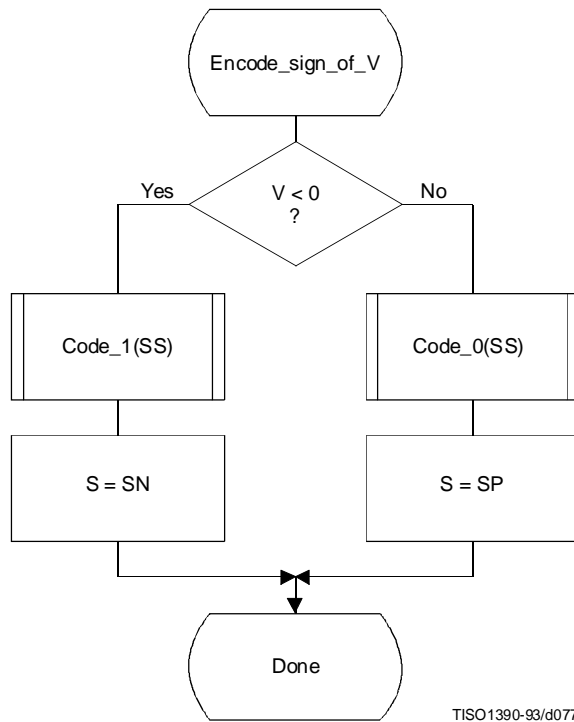
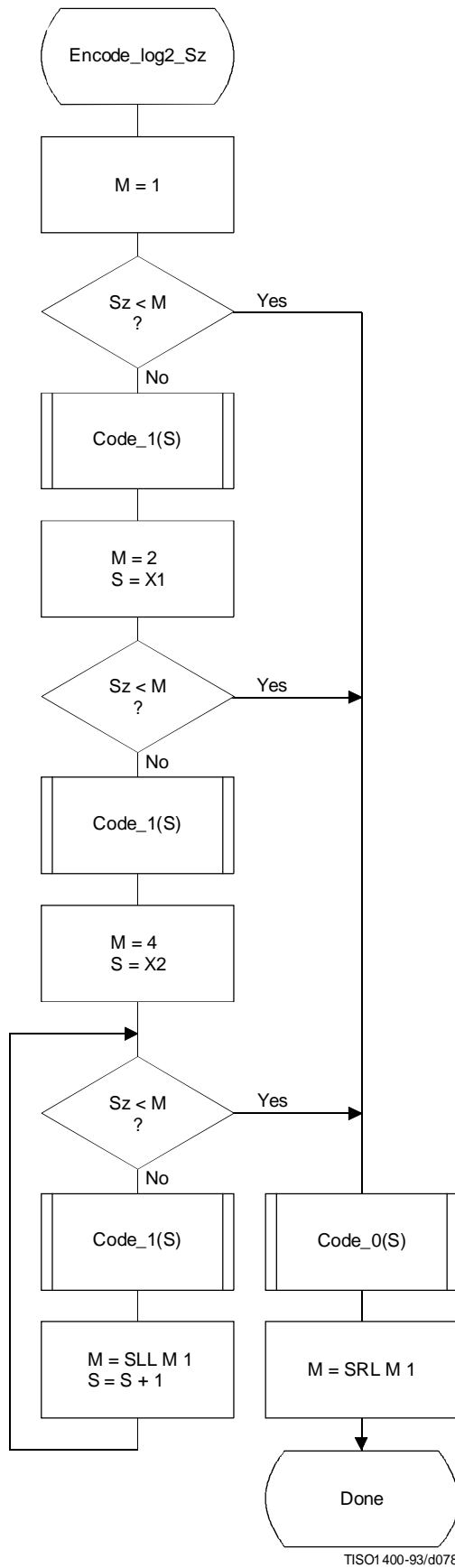


Figure F.7 – Encoding the sign of V

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



TISO1 400-93/d078

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.

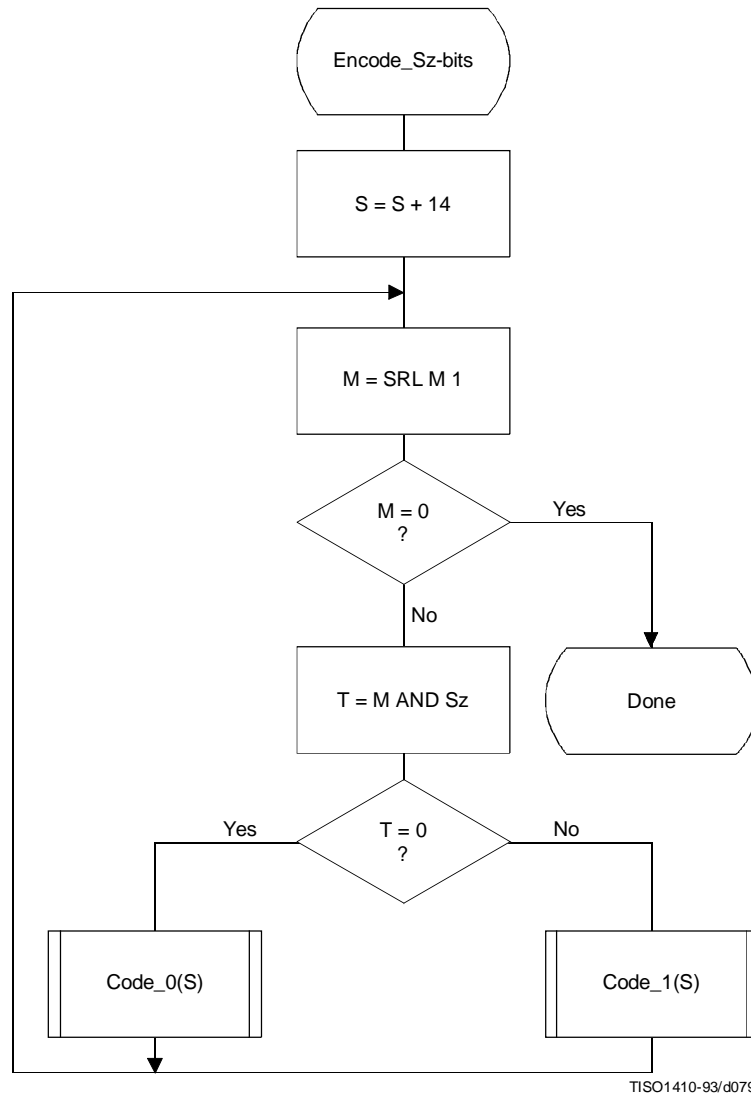


Figure F.9 – Decision sequence to code the magnitude bit pattern

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 Q_e 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, $S_z < 1$; 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 D_a , 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.

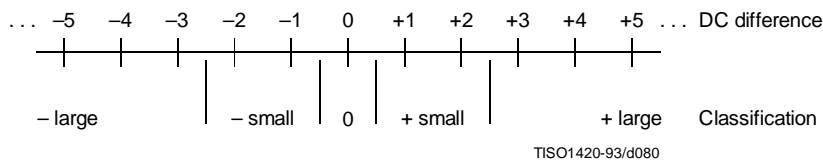


Figure F.10 – Conditioning classification of difference values

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 S_0 . The value of S_0 is given by $DC_Context(D_a)$, which provides a value of 0, 4, 8, 12 or 16, depending on the difference classification of D_a (see F.1.4.4.1.2). The remaining 29 bins, $X_1, \dots, X_{15}, M_2, \dots, M_{15}$, 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 $X2, \dots, X15, M2, \dots, M15$ 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 ($Q_e = X'5A1D'$, $MPS = 0$).

The value of X2 is given by $AC_Context(K)$. This gives $X2 = 189$ when $K \leq K_x$ and $X2 = 217$ when $K > K_x$, where K_x 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 X_n , for example – a single statistics bin, M_n , 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 K_x 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.

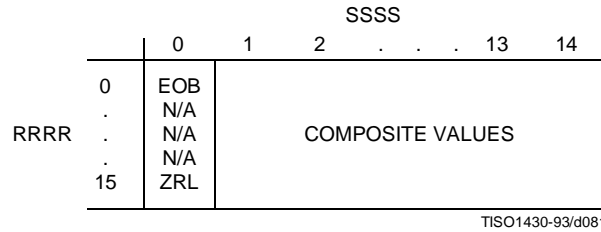


Figure F.11 – Two-dimensional value array for Huffman coding

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:

$$T = \text{DECODE}$$

$$\text{DIFF} = \text{RECEIVE}(T)$$

$$\text{DIFF} = \text{EXTEND}(\text{DIFF}, T)$$

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.

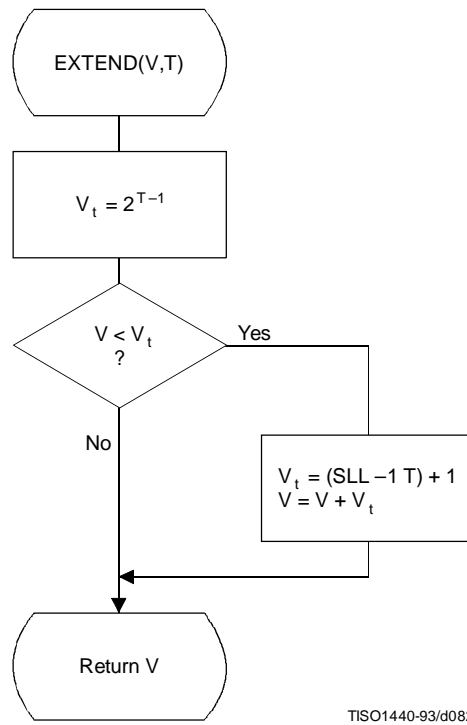
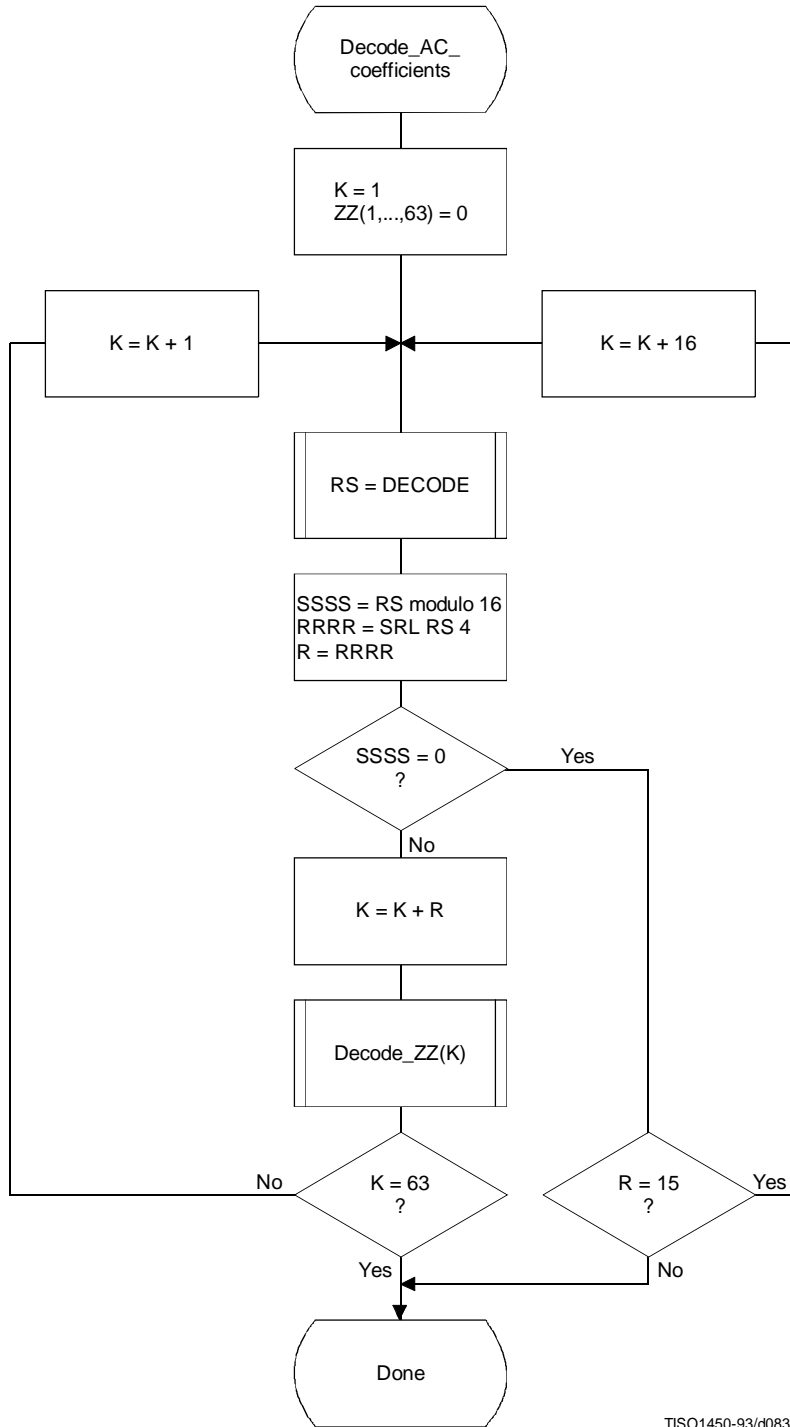


Figure F.12 – Extending the sign bit of a decoded value in V

F.2.2.2 Decoding procedure for AC coefficients

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



TISO1450-93/d083

Figure F.13 – Huffman decoding procedure for AC coefficients

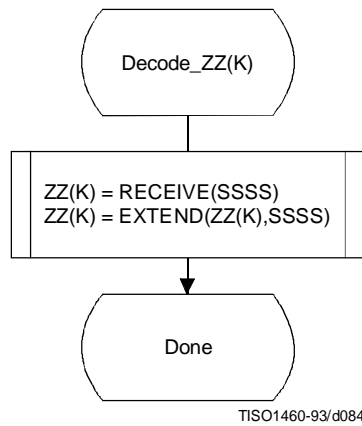


Figure F.14 – Decoding a non-zero AC coefficient

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.

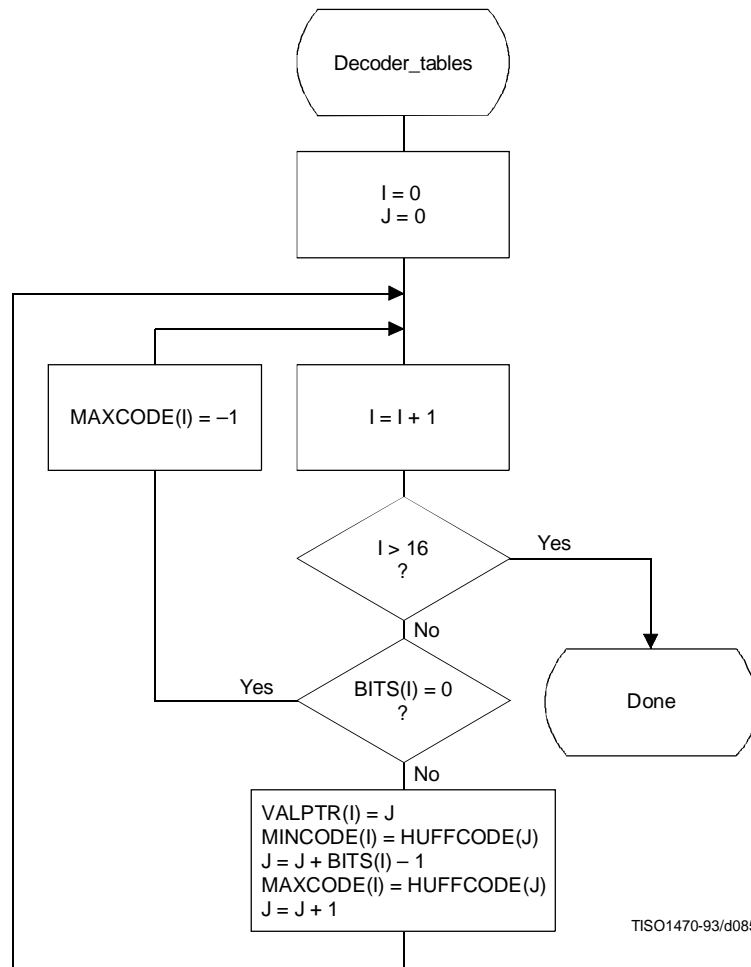


Figure F.15 – Decoder table generation

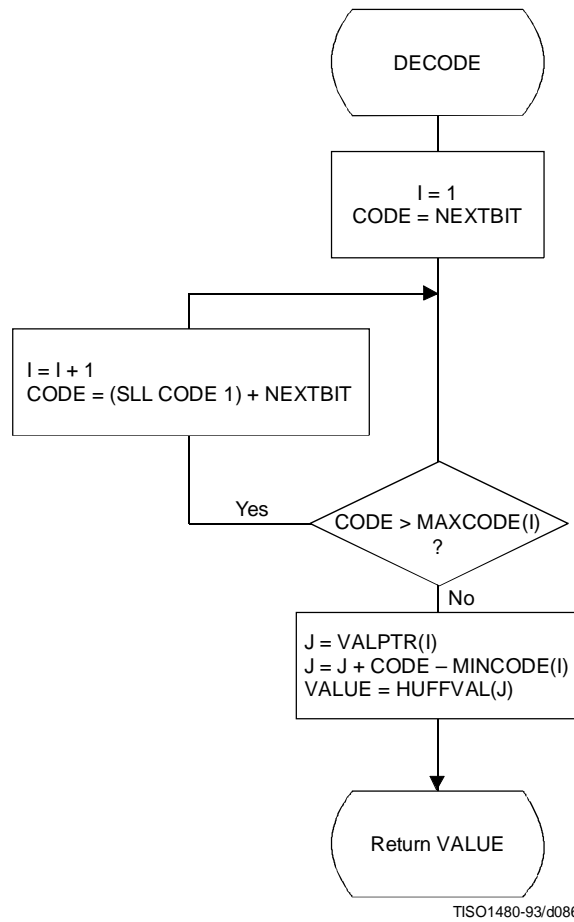


Figure F.16 – Procedure for DECODE

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).

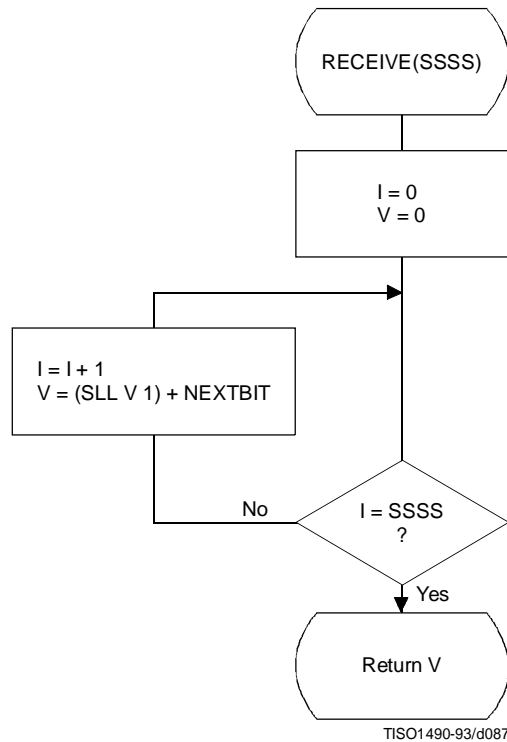


Figure F.17 – Procedure for RECEIVE(SSSS)

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.

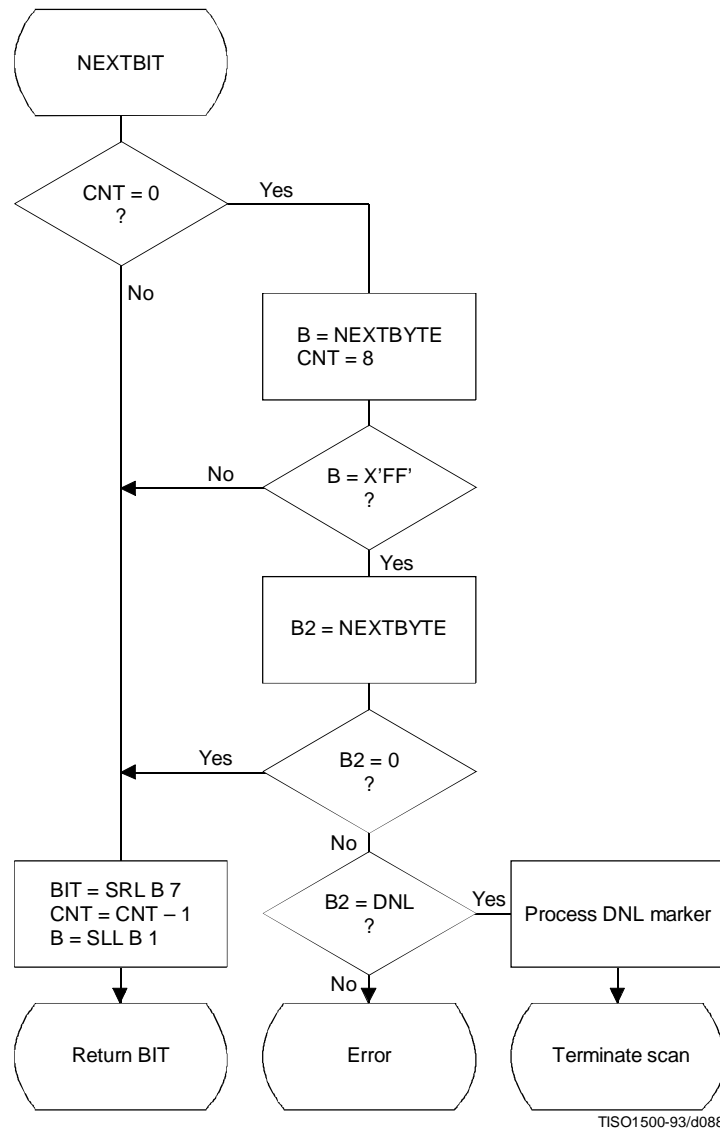


Figure F.18 – Procedure for fetching the next bit of compressed data

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.

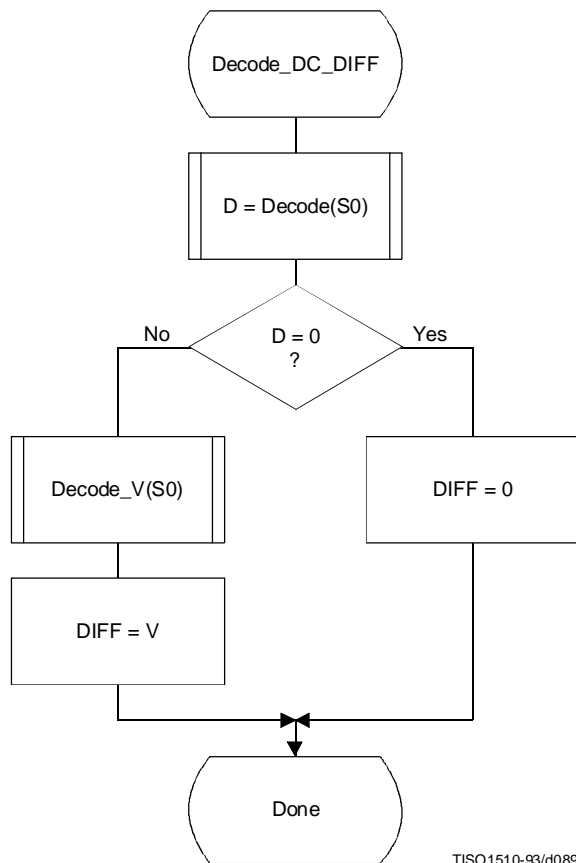


Figure F.19 – Arithmetic decoding of DC difference

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.

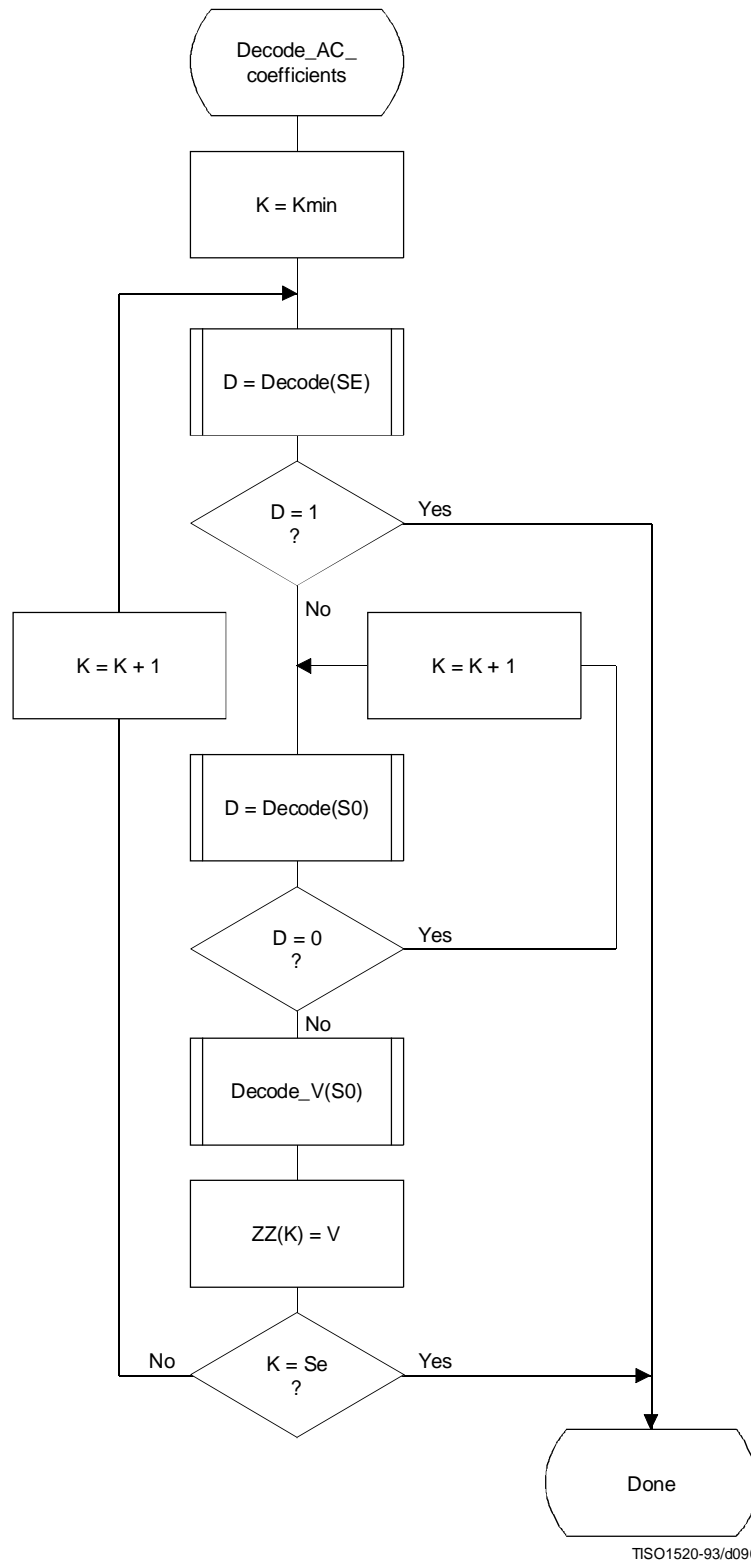


Figure F.20 – Procedure for decoding the AC coefficients

The context-indices used in the AC decoding procedures are defined in Table F.5 (see F.1.4.4.2).

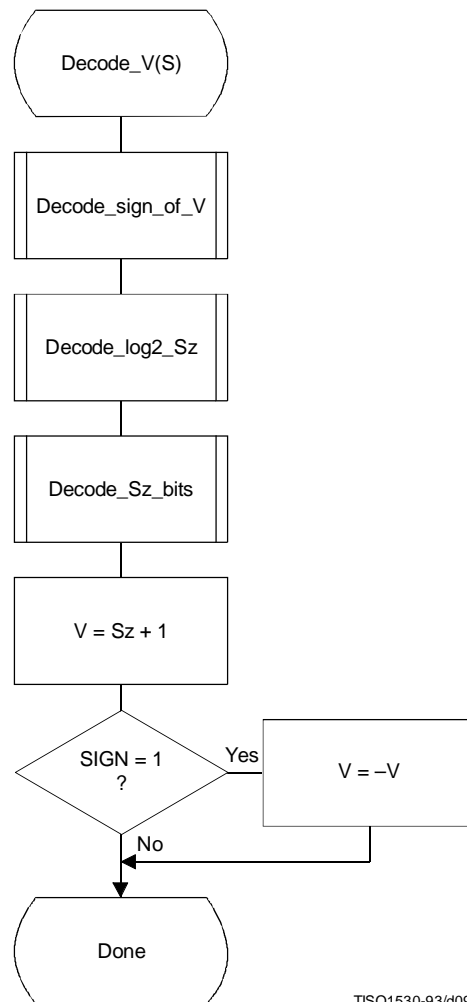
In Figure F.20, K is the index to the zig-zag sequence position. For the sequential scan, $K_{min} = 1$ and $Se = 63$. The decision at the top of the loop is the EOB decision. If the EOB occurs ($D = 1$), the remaining coefficients in the block are set to zero. The inner loop just below the EOB decoding 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.



TISO1530-93/d091

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.

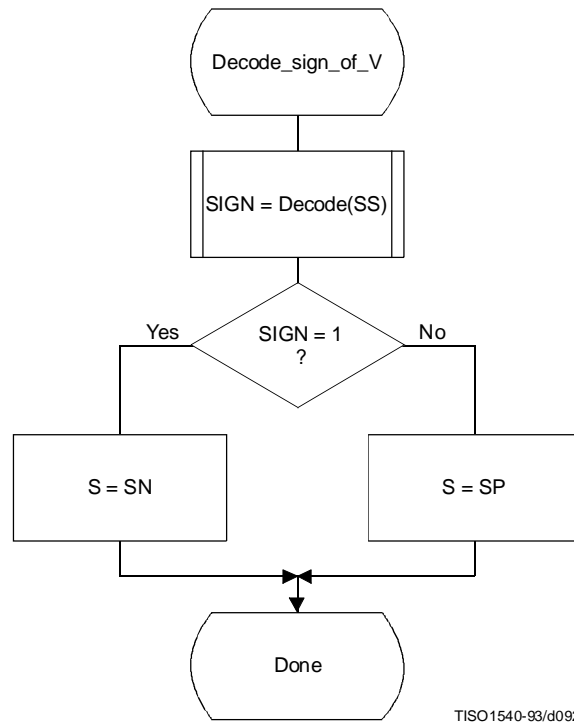


Figure F.22 – Decoding the sign of V

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 S_z .

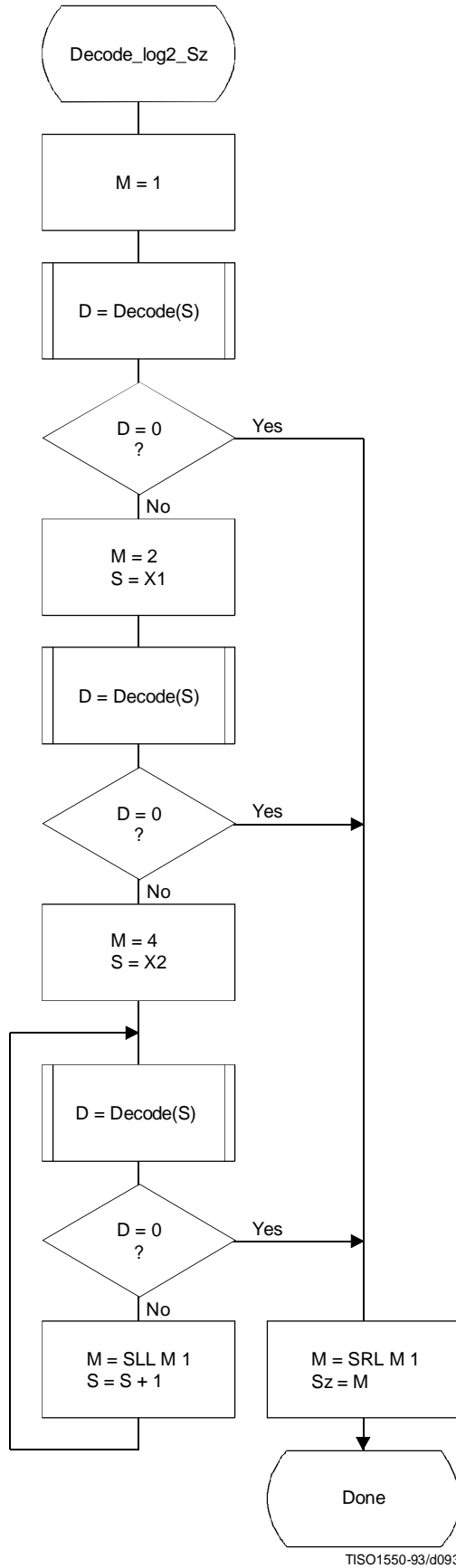


Figure F.23 – Decoding procedure to establish the magnitude category

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.

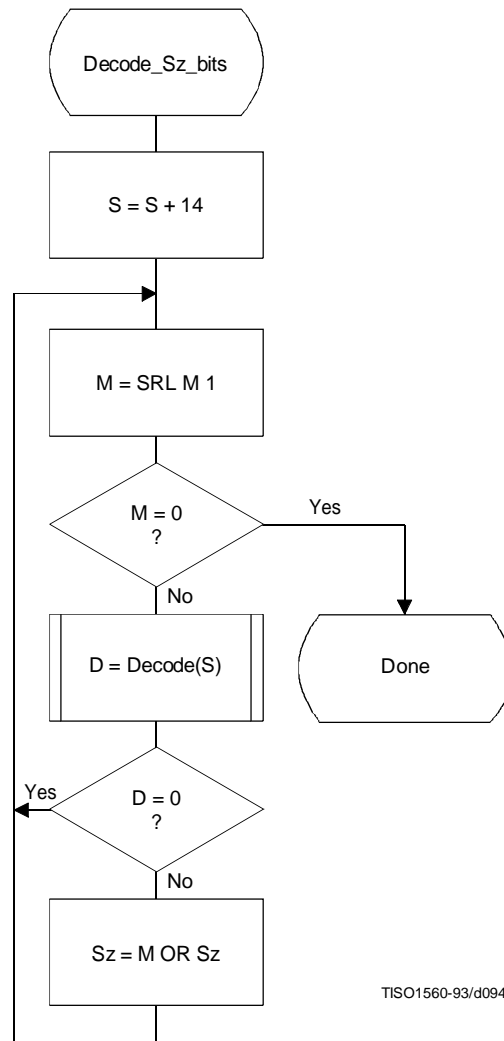


Figure F.24 – Decision sequence to decode the magnitude bit pattern

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.

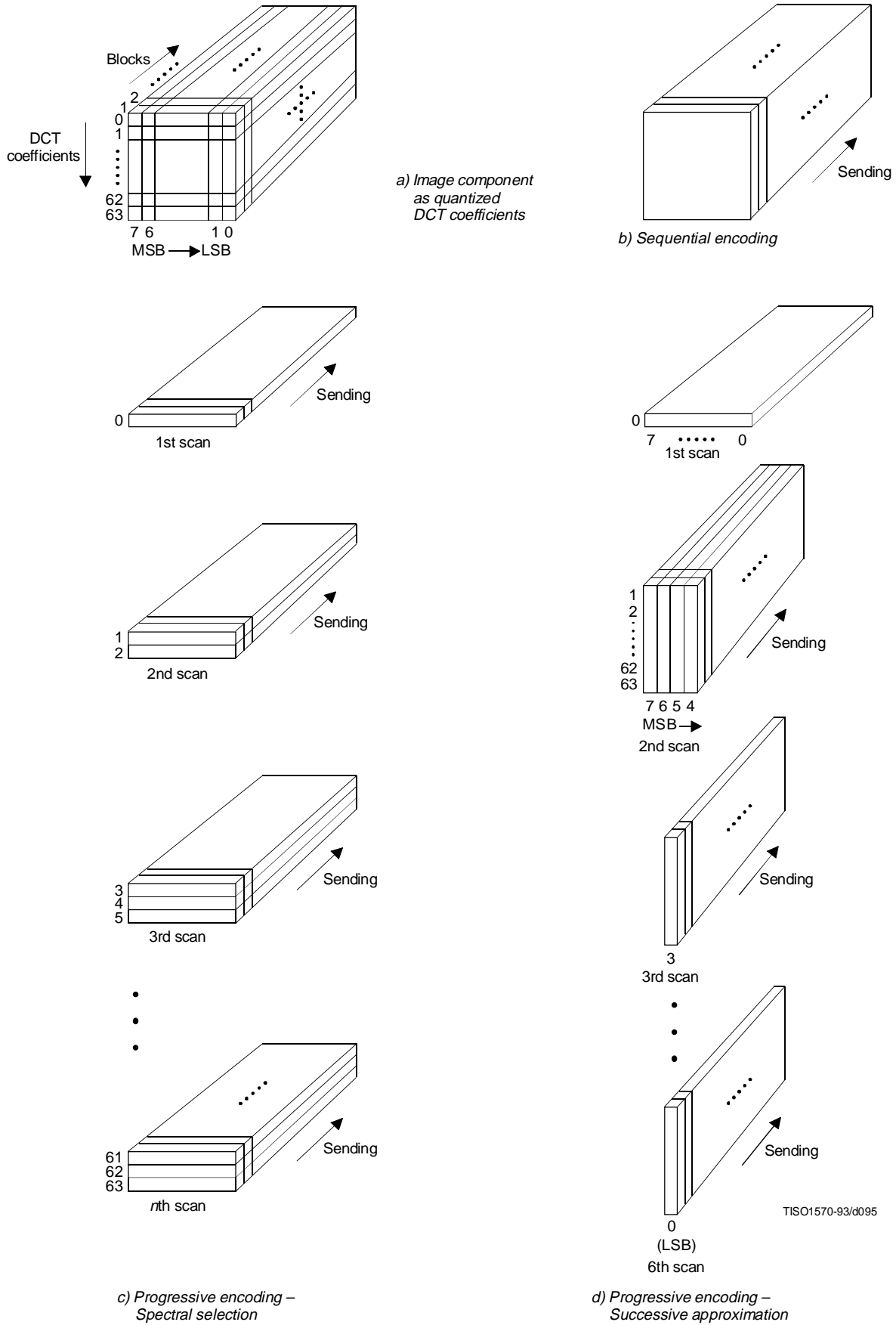


Figure G.1 – Spectral selection and successive approximation progressive processes

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 S_s , S_e , A_h , and A_l (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 A_l specifies the actual point transform, and the high four bits (A_h) – 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, A_h 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 A_l is not zero, the point transform for DC coefficients shall be used to reduce the precision of the DC coefficients. If A_h is zero, the coefficient values (as modified by the point transform) shall be coded, using the procedure described in Annex F. If A_h 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 A_l is not zero, the point transform for AC coefficients shall be used to reduce the precision of the AC coefficients. If A_h 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 A_h 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 A_l 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.

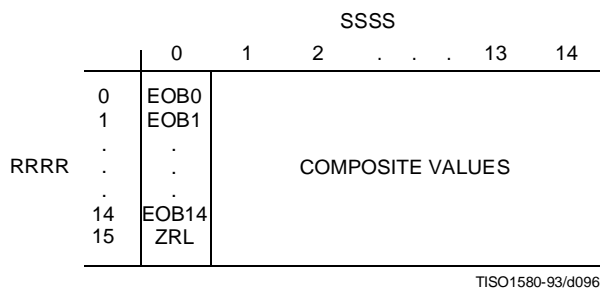


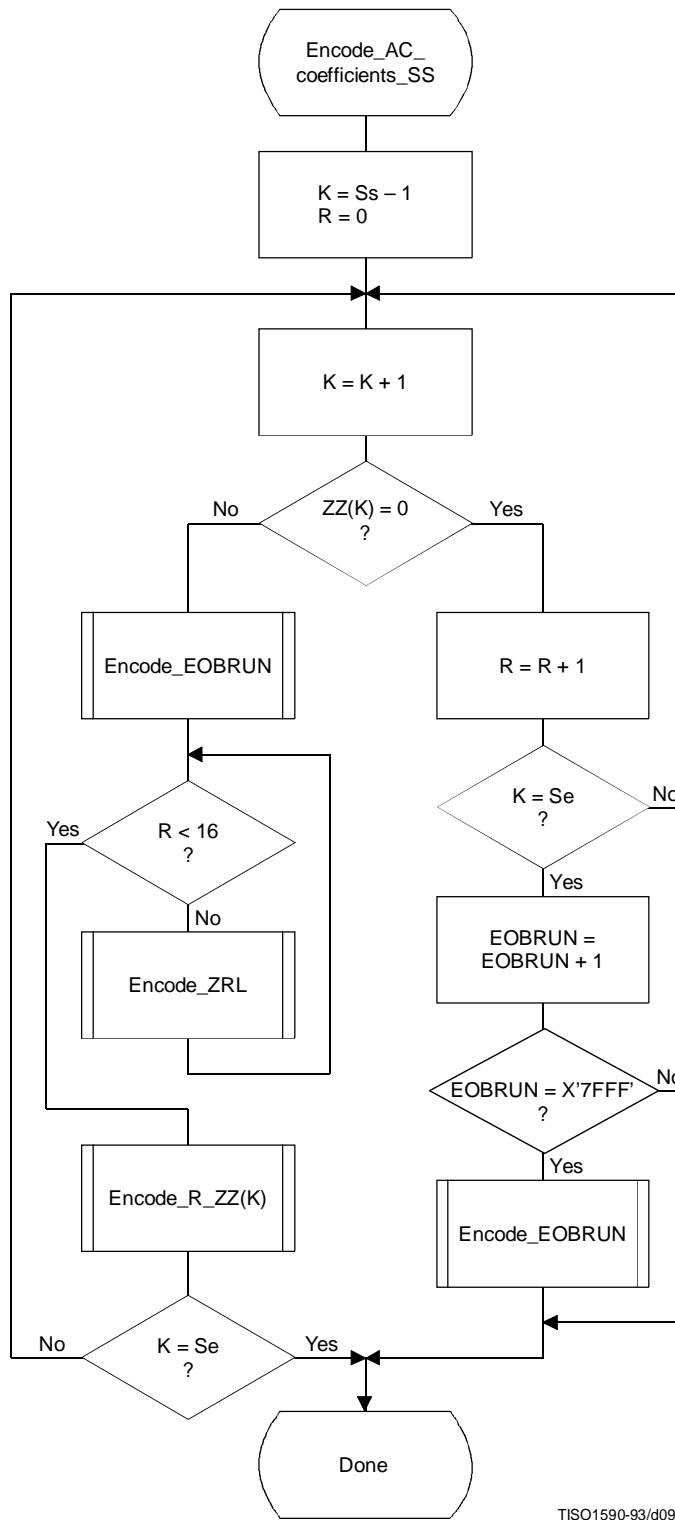
Figure G.2 – Two-dimensional value array for Huffman coding

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.



TISO1590-93/d097

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

In Figure G.3, S_s is the start of spectral selection, S_e 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 A_l (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:

$$ZZ'(K) = \frac{ZZ(K) \times X}{2^{A_l}}$$

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).

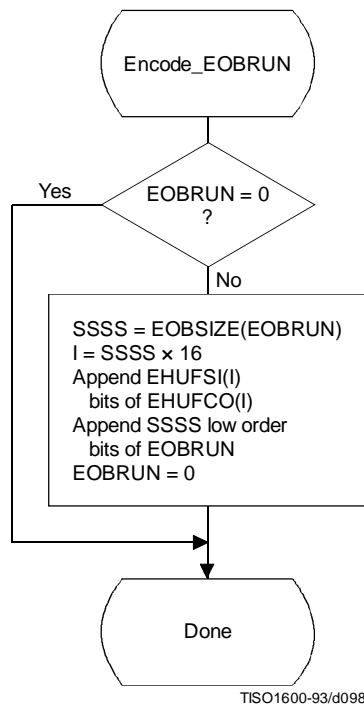


Figure G.4 – Progressive encoding of a non-zero AC coefficient

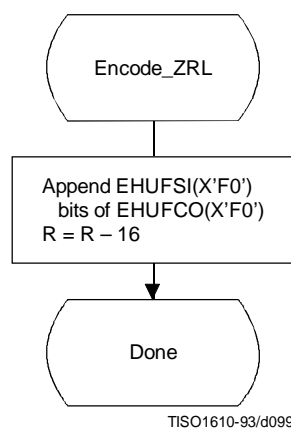


Figure G.5 – Encoding of the run of zero coefficients

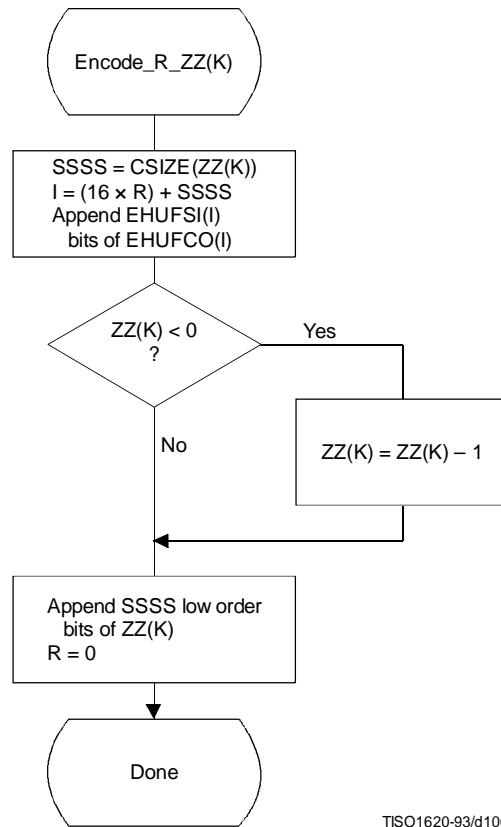


Figure G.6 – Encoding of the zero run and non-zero coefficient

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:

$$\text{HUFFCO(RRRRSSS)} + \text{additional bit (rule a)} + \text{correction bits (rule b)}$$

In addition whenever zero runs are coded with ZRL or EOB_n 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 EOB_n 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 ($Q_e = X'5A1D'$, MPS = 0).

G.1.3.2 Progressive encoding of AC coefficients with arithmetic coding

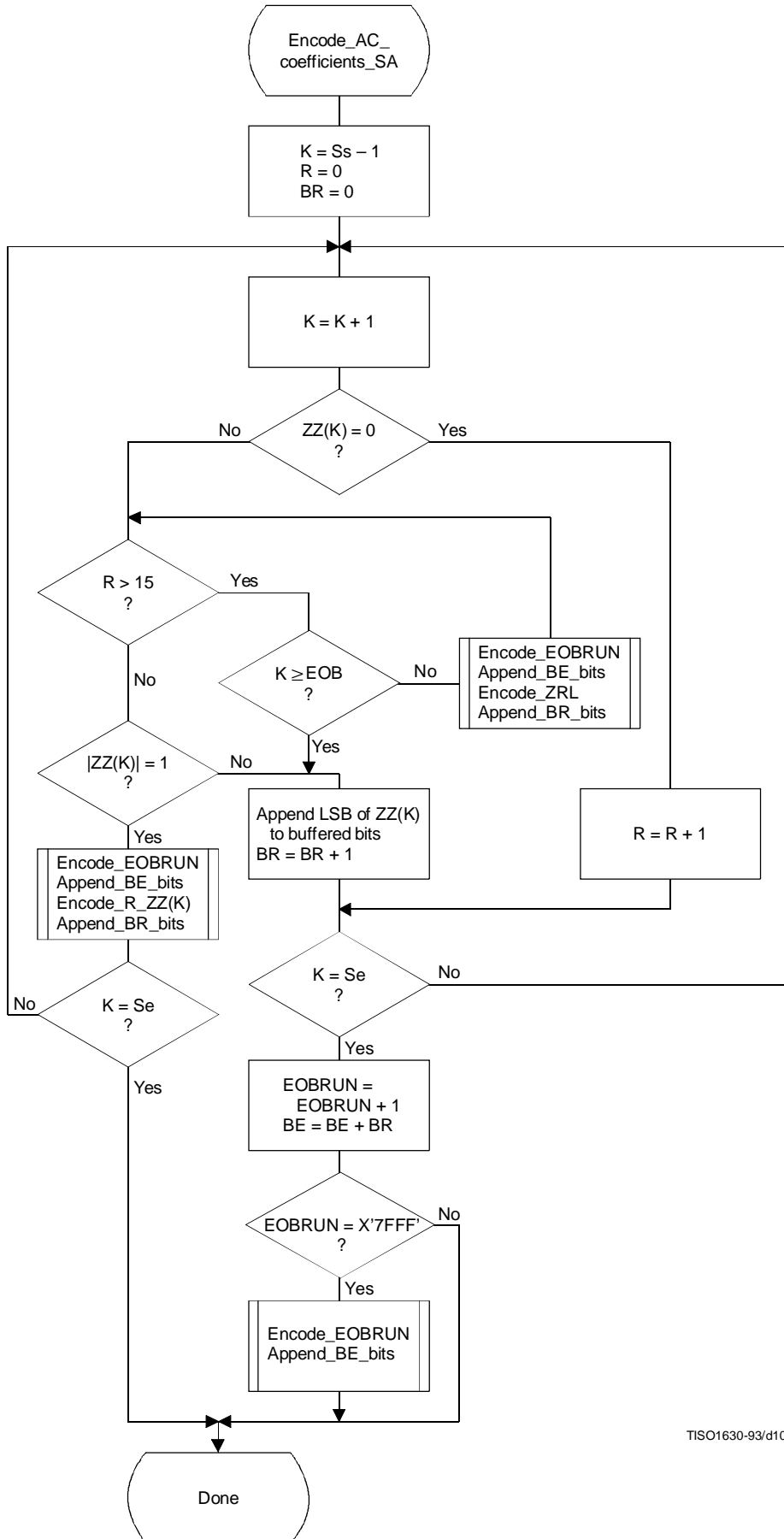
Except for the point transform scaling of the DCT coefficients and the grouping of the coefficients into bands, the first scan(s) of successive approximation is identical to the sequential encoding procedure described in F.1.4. If K_{min} is equated to S_s, the index of the first AC coefficient index 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 K_x is 5. Other values of K_x may be specified using the DAC marker code (Annex B). The following calculation for K_x has proven to give good results for 8-bit precision samples:

$$K_x = K_{min} + SRL (8 + S_e - K_{min}) / 4$$

This expression reduces to the default of K_x = 5 when the band is from index 1 to index 63.



TISO1630-93/d101

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

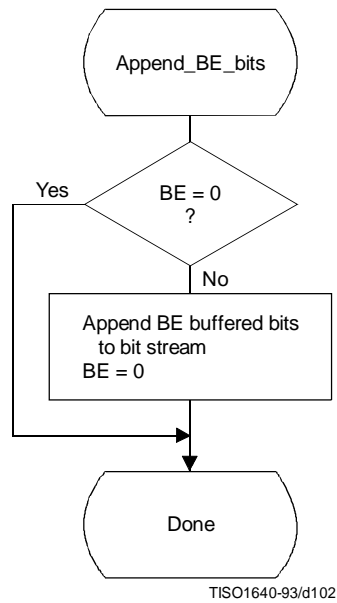


Figure G.8 – Transferring BE buffered bits from buffer to bit stream

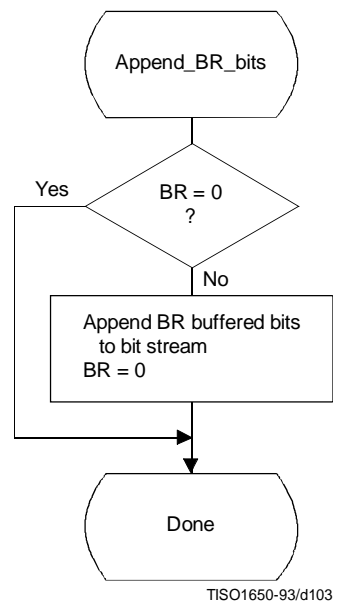


Figure G.9 – Transferring BR buffered bits from buffer to bit stream

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, EOB_x , 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 EOB_x . 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 ($Q_e = X'5A1D'$, $MPS = 0$).

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 ($Q_e = X'5A1D'$, $MPS = 0$).

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.

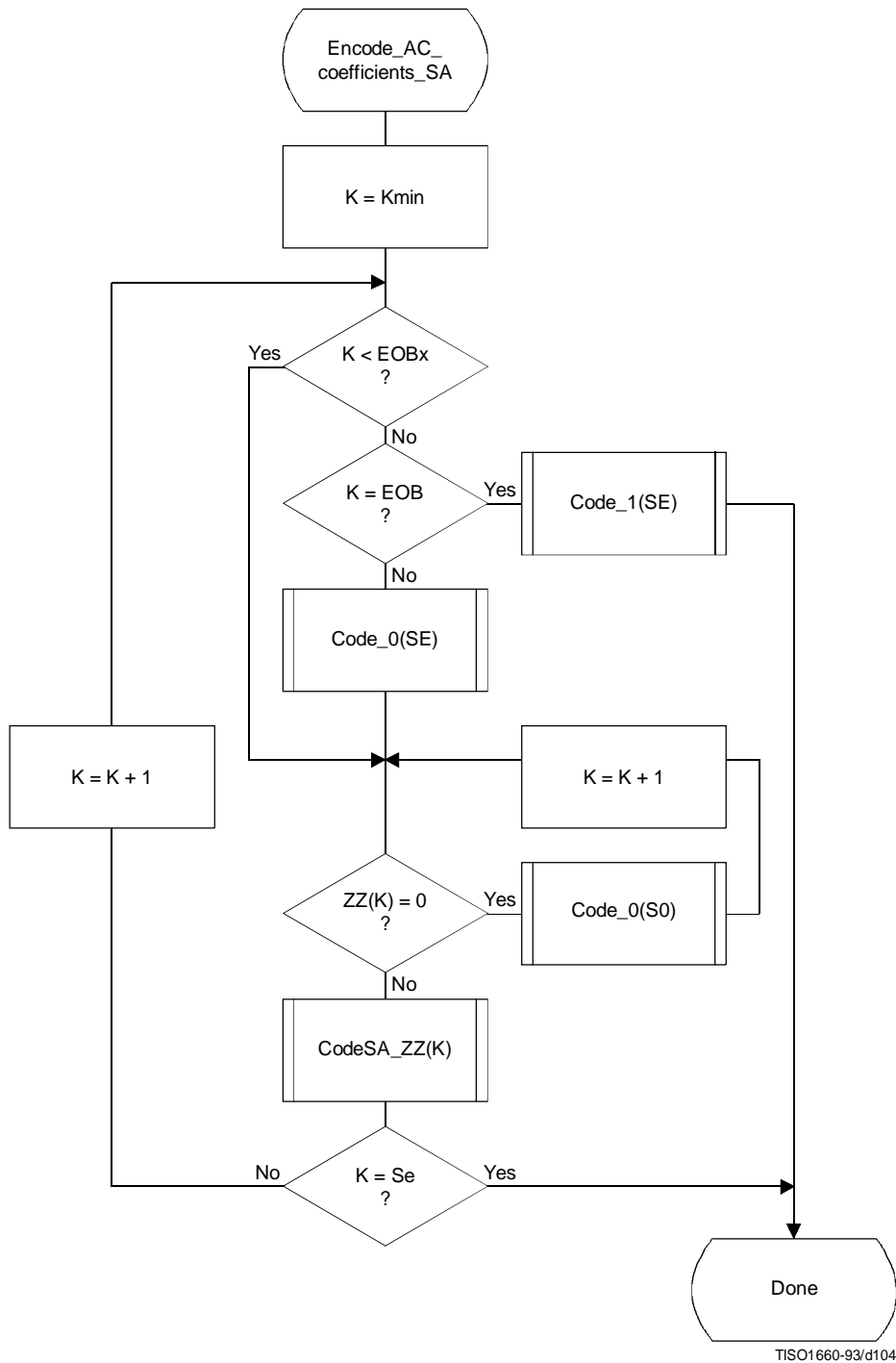


Figure G.10 – Subsequent successive approximation scans for coding of AC coefficients using arithmetic coding

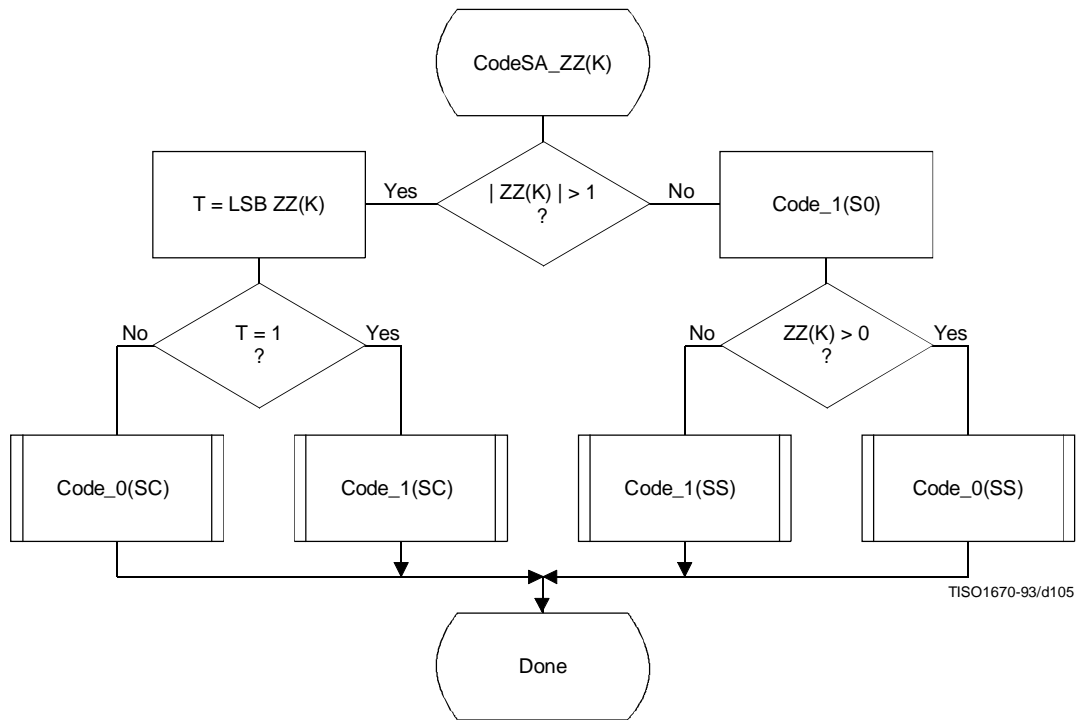


Figure G.11 – Coding non-zero coefficients for subsequent successive approximation scans

Table G.2 – Statistical model for subsequent scans of successive approximation coding of AC coefficient

Context-index	AC coding	Coding decision
SE	$3 \times (K-1)$	$K = \text{EOB}$
S0	$SE + 1$	$V = 0$
SS	Fixed estimate	Sign
SC	$S0 + 1$	$\text{LSB } ZZ(K) = 1$

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.

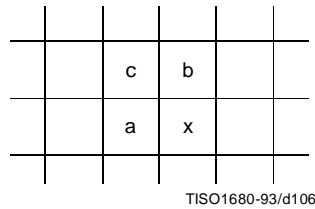


Figure H.1 – Relationship between sample and prediction samples

Define P_x to be the prediction and R_a , R_b , and R_c 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	$P_x = R_a$
2	$P_x = R_b$
3	$P_x = R_c$
4	$P_x = R_a + R_b - R_c$
5	$P_x = R_a + ((R_b - R_c)/2)^{a)}$
6	$P_x = R_b + ((R_a - R_c)/2)^{a)}$
7	$P_x = (R_a + R_b)/2$
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 R_a) 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 R_b) 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-P_t-1} , where P_t 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 R_a and R_b 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)				
		0	+S	-S	+L	-L
Difference to left (position a)	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.

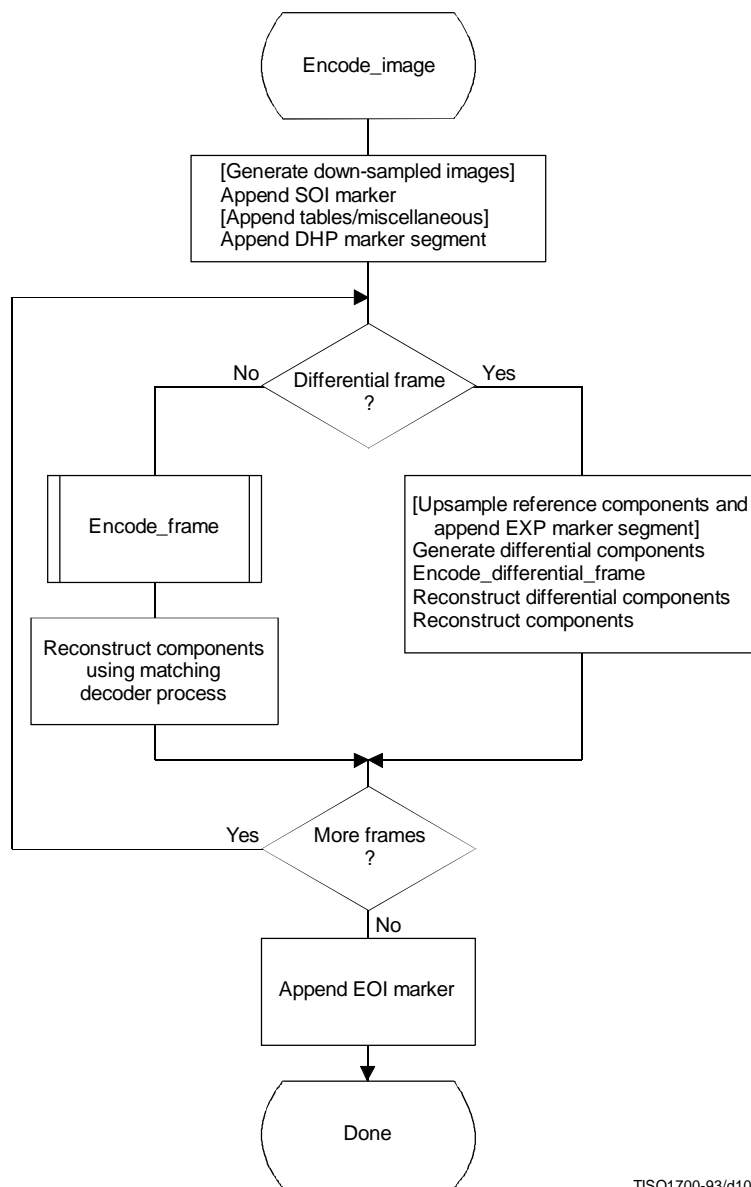
NOTE – There is **no requirement** in this Specification that any encoder or decoder which embodies one of the above-named processes shall implement the procedures in precisely the manner specified by the flow charts in this annex. It is necessary only that an encoder or decoder implement the **function** 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.



TISO1700-93/d108

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

In Figure J.1 procedures in brackets shall be performed whenever the particular hierarchical encoding sequence being followed requires them.

In the hierarchical mode the define-hierarchical-progression (DHP) marker segment shall be placed in the compressed image data before the first start-of-frame. The DHP segment is used to signal the size of the image components of the completed image. The syntax of the DHP segment is specified in Annex B.

The first frame for each component or group of components in a hierarchical process shall be encoded by a non-differential frame. Differential frames shall then be used to encode the two's complement differences between source input components (possibly downsampled) and the reference components (possibly upsampled). The reference components are reconstructed components created by previous frames in the hierarchical process. For either differential or non-differential frames, reconstructions of the components shall be generated if needed as reference components for a subsequent frame in the hierarchical process.

Resolution changes may occur between hierarchical frames in a hierarchical process. These changes occur if downsampling filters are used to reduce the spatial resolution of some or all of the components of the source image. When the resolution of a reference component does not match the resolution of the component input to a differential frame, an upsampling filter shall be used to increase the spatial resolution of the reference component. The EXP marker segment shall be added to the compressed image data before the start-of-frame whenever upsampling of a reference component is required. No more than one EXP marker segment shall precede a given frame.

Any of the marker segments allowed before a start-of-frame for the encoding process selected may be used before either non-differential or differential frames.

For 16-bit input precision (lossless encoder), the differential components which are input to a differential frame are calculated modulo 2^{16} . The reconstructed components calculated from the reconstructed differential components are also calculated modulo 2^{16} .

If a hierarchical encoding process uses a DCT encoding process for the first frame, all frames in the hierarchical process except for the final frame for each component shall use the DCT encoding processes defined in either Annex F or Annex G, or the modified DCT encoding processes defined in this annex. The final frame may use a modified lossless process defined in this annex.

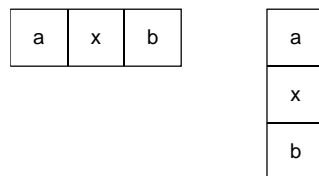
If a hierarchical encoding process uses a lossless encoding process for the first frame, all frames in the hierarchical process shall use a lossless encoding process defined in Annex H, or a modified lossless process defined in this annex.

J.1.1.1 Downsampling filter

The downsampled components are generated using a downsampling filter that is not specified in this Specification. This filter should, however, be consistent with the upsampling filter. An example of a downsampling filter is provided in K.5.

J.1.1.2 Upsampling filter

The upsampling filter increases the spatial resolution by a factor of two horizontally, vertically, or both. Bi-linear interpolation is used for the upsampling filter, as illustrated in Figure J.2.



TISO1710-93/d109

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.

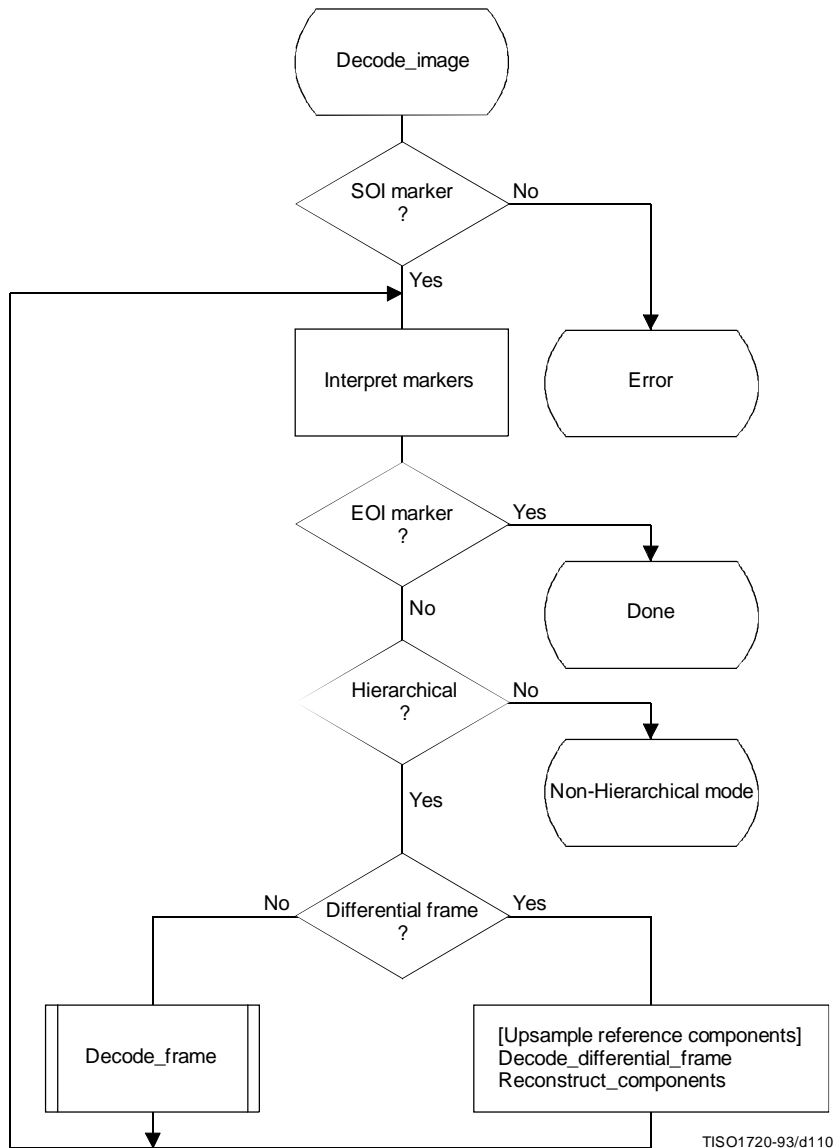


Figure J.3 – Hierarchical control procedure for decoding an image

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.

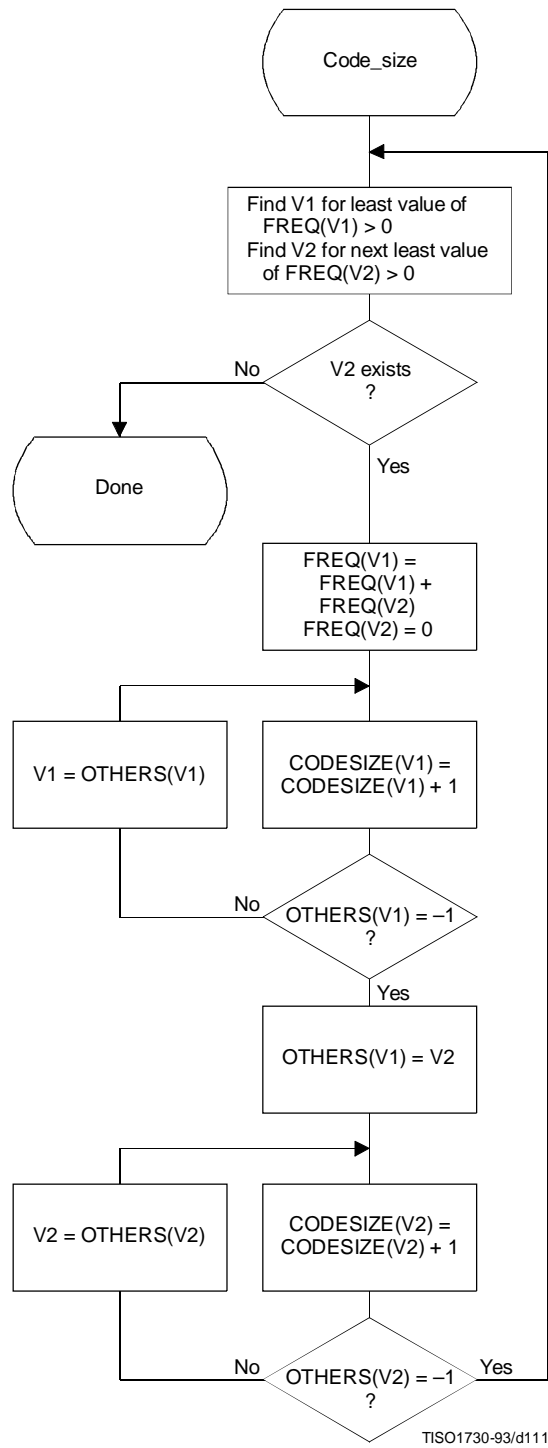
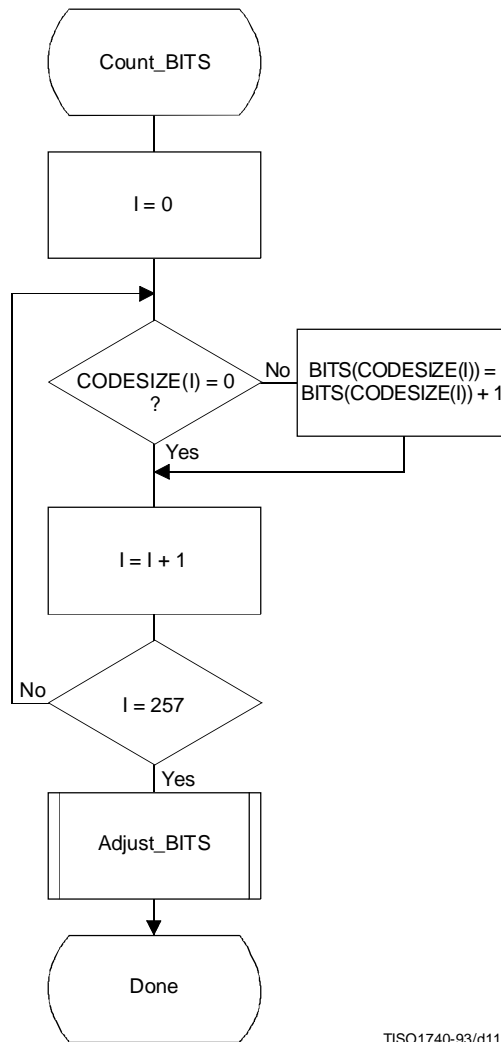


Figure K.1 – Procedure to find Huffman code sizes

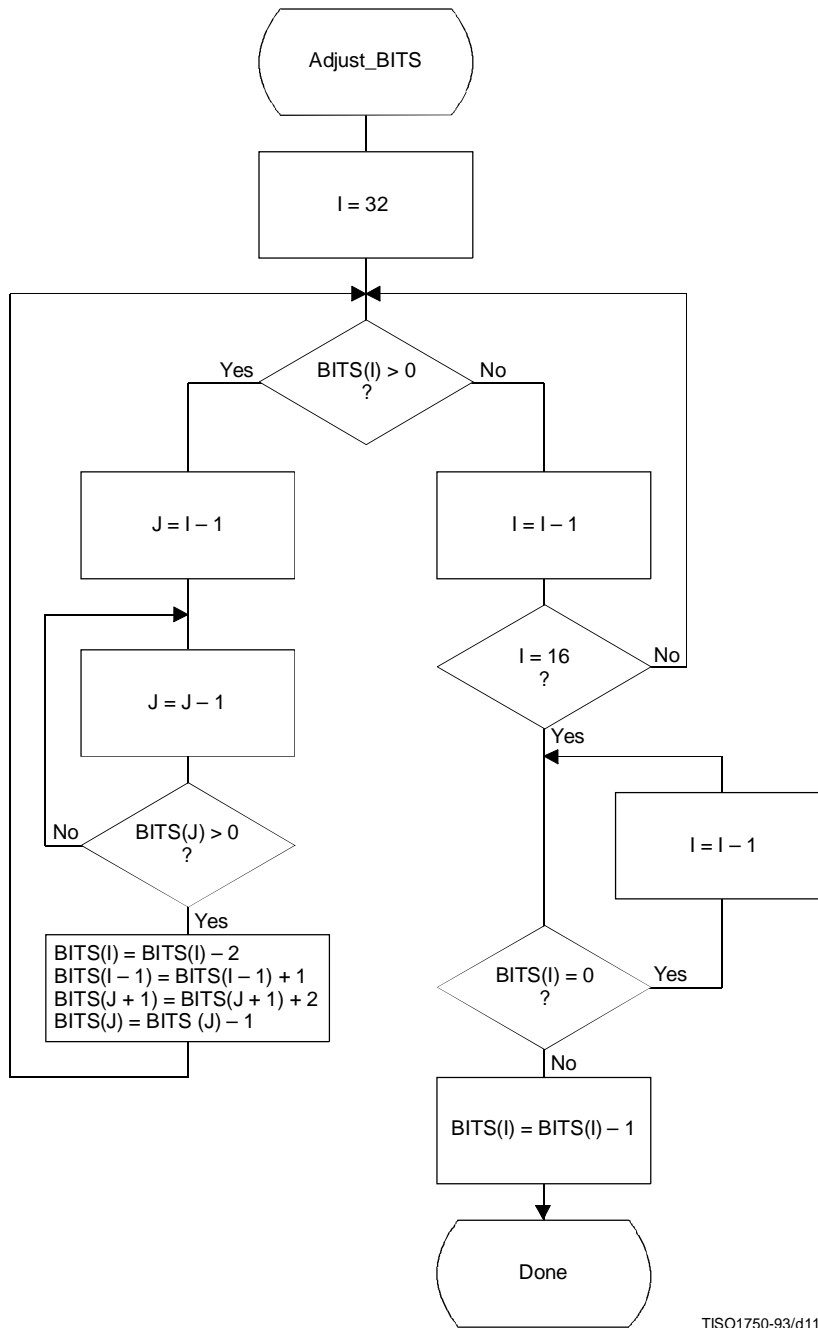
Once the code lengths for each symbol have been obtained, the number of codes of each length is obtained using the procedure in Figure K.2. The count for each size is contained in the list, BITS. The counts in BITS are zero at the start of the procedure. The procedure assumes that the probabilities are 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).



TISO1740-93/d112

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.

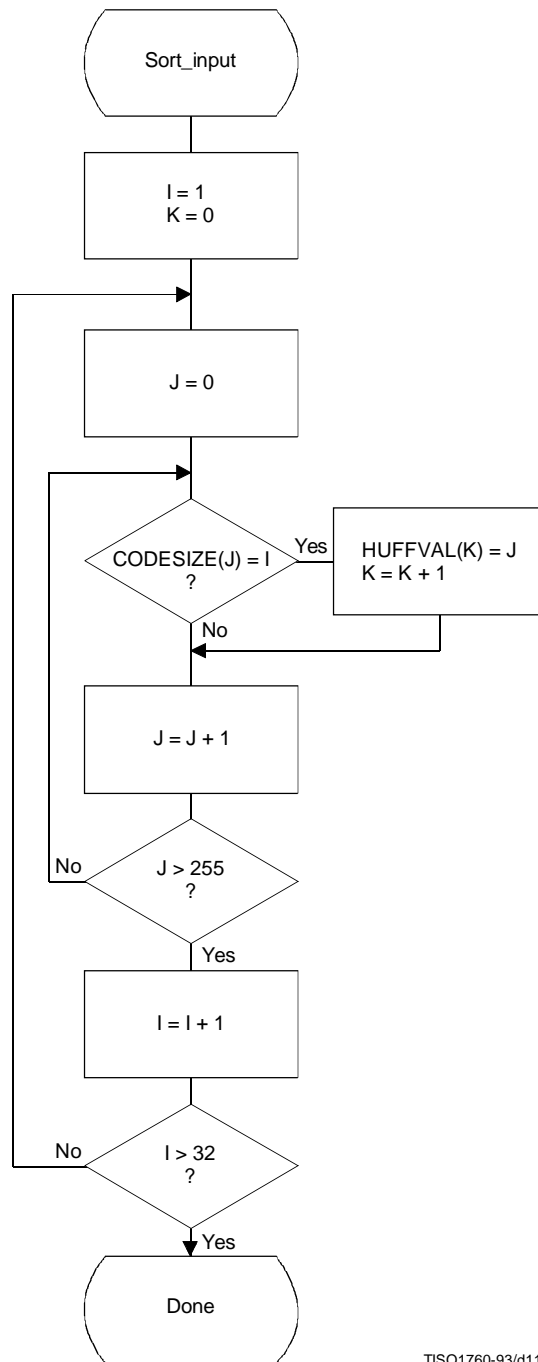


TISO1750-93/d113

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.



TISO1760-93/d114

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	111111110010110
4/4	16	111111110010111
4/5	16	111111110011000
4/6	16	111111110011001
4/7	16	111111110011010
4/8	16	111111110011011
4/9	16	111111110011100
4/A	16	111111110011101
5/1	7	1111010
5/2	11	11111110111
5/3	16	111111110011110
5/4	16	111111110011111
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	111111110110110
8/4	16	111111110110111
8/5	16	111111110111000
8/6	16	111111110111001
8/7	16	111111110111010
8/8	16	111111110111011
8/9	16	111111110111100
8/A	16	111111110111101
9/1	9	11111001
9/2	16	111111110111110
9/3	16	111111110111111
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	11111010
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	111111001
B/2	16	111111111010000
B/3	16	111111111010001
B/4	16	111111111010010
B/5	16	111111111010011
B/6	16	111111111010100
B/7	16	111111111010101
B/8	16	111111111010110
B/9	16	111111111010111
B/A	16	111111111011000
C/1	10	111111010
C/2	16	111111111011001
C/3	16	111111111011010
C/4	16	111111111011011

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	111111111100000
C/A	16	111111111100001
D/1	11	1111111000
D/2	16	111111111100010
D/3	16	111111111100011
D/4	16	111111111100100
D/5	16	111111111100101
D/6	16	111111111100110
D/7	16	111111111100111
D/8	16	111111111101000
D/9	16	111111111101001
D/A	16	111111111101010
E/1	16	111111111101011
E/2	16	111111111101100
E/3	16	111111111101101
E/4	16	111111111101110
E/5	16	111111111101111
E/6	16	111111111100000
E/7	16	111111111100001
E/8	16	111111111100010
E/9	16	111111111100011
E/A	16	111111111101010
F/0 (ZRL)	11	1111111001
F/1	16	111111111110101
F/2	16	111111111110110
F/3	16	111111111110111
F/4	16	111111111110000
F/5	16	111111111110001
F/6	16	111111111110010
F/7	16	111111111110011
F/8	16	111111111110100
F/9	16	111111111110101
F/A	16	111111111110110

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	111111110010111
4/4	16	111111110011000
4/5	16	111111110011001
4/6	16	111111110011010
4/7	16	111111110011011
4/8	16	111111110011100
4/9	16	111111110011101
4/A	16	111111110011110
5/1	6	111011
5/2	10	1111111001
5/3	16	111111110011111
5/4	16	111111110100000
5/5	16	111111110100001
5/6	16	111111110100010
5/7	16	111111110100011
5/8	16	111111110100100
5/9	16	111111110100101
5/A	16	111111110100110
6/1	7	1111001
6/2	11	11111110111
6/3	16	111111110100111
6/4	16	111111110101000
6/5	16	111111110101001
6/6	16	111111110101010
6/7	16	111111110101011
6/8	16	111111110101100
6/9	16	111111110101101
6/A	16	111111110101110
7/1	7	1111010
7/2	11	11111111000
7/3	16	111111110101111
7/4	16	111111110110000
7/5	16	111111110110001
7/6	16	111111110110010
7/7	16	111111110110011
7/8	16	111111110110100
7/9	16	111111110110101
7/A	16	111111110110110
8/1	8	11111001
8/2	16	111111110110111
8/3	16	111111110111000

Table K.6 (sheet 3 of 4)

Run/Size	Code length	Code word
8/4	16	111111110111001
8/5	16	111111110111010
8/6	16	111111110111011
8/7	16	111111110111100
8/8	16	111111110111101
8/9	16	111111110111110
8/A	16	111111110111111
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	1111111100000
E/2	16	1111111111101101
E/3	16	1111111111101110
E/4	16	1111111111101111
E/5	16	1111111111100000
E/6	16	1111111111100001
E/7	16	1111111111100010
E/8	16	1111111111100011
E/9	16	1111111111101000
E/A	16	1111111111101001
F/0 (ZRL)	10	1111111010
F/1	15	11111111000011
F/2	16	1111111111101110
F/3	16	1111111111101111
F/4	16	1111111111110000
F/5	16	1111111111110001
F/6	16	1111111111110010
F/7	16	1111111111110011
F/8	16	1111111111111000
F/9	16	1111111111111001
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 AAAAAAAAA 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	CE	5A1D	0000	00000000	11	0		
2	0	0		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		55
113	1	0		415E	8C72	0005738E	8	0		
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	CDDE	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	001ECEFO	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	023AEB44	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	CE	5A1D	0000	655B0000	0	65 5B
2	0	0		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	5CACCC00	3	
197	0	1		415E	C7F2	B9598000	2	
198	1	1	CE	4639	82BC	658B0000	1	
199	0	1		415E	8C72	52100000	0	
200	0	1	CE	4639	82BC	0DF8E000	7	70

Table K.8 – Decoder test sequence (sheet 6 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	B
201	1	1		4B85	F20C	37E38000	5	
202	1	1		4B85	A687	37E38000	5	
203	1	1		4639	B604	6FC70000	4	
204	0	1		415E	DF96	DF8E0000	3	
205	1	1	CE	4639	82BC	82AC0000	2	
206	0	1		415E	8C72	8C520000	1	
207	1	1	CE	4639	82BC	827C0000	0	
208	0	1		415E	8C72	8BF31C00	7	8E
209	1	1	CE	4639	82BC	81BE3800	6	
210	0	1		415E	8C72	8A767000	5	
211	1	1	CE	4639	82BC	7EC4E000	4	
212	0	1		415E	8C72	8483C000	3	
213	1	1	CE	4639	82BC	72DF8000	2	
214	0	1		415E	8C72	6CB90000	1	
215	1	1	CE	4639	82BC	434A0000	0	
216	1	1		415E	8C72	0D8F9600	7	CB
217	1	1		3C3D	9628	1B1F2C00	6	
218	1	1		375E	B3D6	363E5800	5	
219	1	1		32B4	F8F0	6C7CB000	4	
220	1	1		32B4	C63C	6C7CB000	4	
221	0	1		32B4	9388	6C7CB000	4	
222	1	1		3C3D	CAD0	2EA2C000	2	
223	1	1		3C3D	8E93	2EA2C000	2	
224	1	1		375E	A4AC	5D458000	1	
225	0	1		32B4	DA9C	BA8B0000	0	
226	1	1		3C3D	CAD0	4A8F0000	6	C0
227	1	1		3C3D	8E93	4A8F0000	6	
228	0	1		375E	A4AC	951E0000	5	
229	1	1		3C3D	DD78	9F400000	3	
230	0	1		3C3D	A13B	9F400000	3	
231	0	1		415E	F0F4	E9080000	1	
232	1	1	CE	4639	82BC	72E40000	0	
233	0	1		415E	8C72	6CC3EC00	7	F6
234	1	1	CE	4639	82BC	435FD800	6	
235	1	1		415E	8C72	0DB9B000	5	
236	1	1		3C3D	9628	1B736000	4	
237	1	1		375E	B3D6	36E6C000	3	
238	1	1		32B4	F8F0	6DCD8000	2	
239	1	1		32B4	C63C	6DCD8000	2	
240	0	1		32B4	9388	6DCD8000	2	

Table K.8 – Decoder test sequence (sheet 7 of 7)

EC	D	MPS	CX	Qe (hexadecimal)	A (hexadecimal)	C (hexadecimal)	CT	B
241	1	1		3C3D	CAD0	33E60000	0	
242	1	1		3C3D	8E93	33E60000	0	
Marker detected: zero byte fed to decoder								
243	1	1		375E	A4AC	67CC0000	7	
244	0	1		32B4	DA9C	CF980000	6	
245	0	1		3C3D	CAD0	9EC00000	4	
246	1	1		415E	F0F4	40B40000	2	
247	1	1		415E	AF96	40B40000	2	
248	1	1		3C3D	DC70	81680000	1	
249	0	1		3C3D	A033	81680000	1	
Marker detected: zero byte fed to decoder								
250	1	1		415E	F0F4	75C80000	7	
251	0	1		415E	AF96	75C80000	7	
252	0	1	CE	4639	82BC	0F200000	6	
253	1	1		4B85	F20C	3C800000	4	
254	1	1		4B85	A687	3C800000	4	
255	0	1		4639	B604	79000000	3	
256	0	1	CE	4B85	8C72	126A0000	2	

K.5 Low-pass downsampling filters for hierarchical coding

In this section simple examples are given of downsampling filters which are compatible with the upsampling filter defined in J.1.1.2.

Figure K.5 shows the weighting of neighbouring samples for simple one-dimensional horizontal and vertical low-pass filters. The output of the filter must be normalized by the sum of the neighbourhood weights.

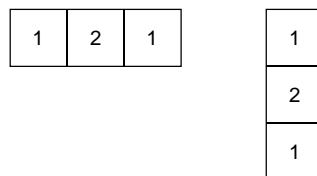


Figure K.5 – Low-pass filter example

The centre sample in Figure K.5 should be aligned with the left column or top line of the high resolution image when calculating the left column or top line of the low resolution image. Sample values which are situated outside of the image boundary are replicated from the sample values at the boundary to provide missing edge values.

If the image being downsampled has an odd width or length, the odd dimension is increased by 1 by sample replication on the right edge or bottom line before downsampling.

K.6 Domain of applicability of DCT and spatial coding techniques

The DCT coder is intended for lossy coding in a range from quite visible loss to distortion well below the threshold for visibility. However in general, DCT-based processes cannot be used for true lossless coding.

The lossless coder is intended for completely lossless coding. The lossless coding process is significantly less effective than the DCT-based processes for distortions near and above the threshold of visibility.

The point transform of the input to the lossless coder permits a very restricted form of lossy coding with the “lossless” coder. (The coder is still lossless after the input point transform.) Since the DCT is intended for lossy coding, there may be some confusion about when this alternative lossy technique should be used.

Lossless coding with a point transformed input is intended for applications which cannot be addressed by DCT coding techniques. Among these are

- true lossless coding to a specified precision;
- lossy coding with precisely defined error bounds;
- hierarchical progression to a truly lossless final stage.

If lossless coding with a point transformed input is used in applications which can be met effectively by DCT coding, the results will be significantly less satisfactory. For example, distortion in the form of visible contours usually appears when precision of the luminance component is reduced to about six bits. For normal image data, this occurs at bit rates well above those for which the DCT gives outputs which are visually indistinguishable from the source.

K.7 Domain of applicability of the progressive coding modes of operation

Two very different progressive coding modes of operation have been defined, progressive coding of the DCT coefficients and hierarchical progression. Progressive coding of the DCT coefficients has two complementary procedures, spectral selection and successive approximation. Because of this diversity of choices, there may be some confusion as to which method of progression to use for a given application.

K.7.1 Progressive coding of the DCT

In progressive coding of the DCT coefficients two complementary procedures are defined for decomposing the 8×8 DCT coefficient array, spectral selection and successive approximation. Spectral selection partitions zig-zag array of DCT coefficients into “bands”, one band being coded in each scan. Successive approximation codes the coefficients with reduced precision in the first scan; in each subsequent scan the precision is increased by one bit.

A single forward DCT is calculated for these procedures. When all coefficients are coded to full precision, the DCT is the same as in the sequential mode. Therefore, like the sequential DCT coding, progressive coding of DCT coefficients is intended for applications which need very good compression for a given level of visual distortion.

The simplest progressive coding technique is spectral selection; indeed, because of this simplicity, some applications may choose – despite the limited progression that can be achieved – to use only spectral selection. Note, however, that the absence of high frequency bands typically leads – for a given bit rate – to a significantly lower image quality in the intermediate stages than can be achieved with the more general progressions. The net coding efficiency at the completion of the final stage is typically comparable to or slightly less than that achieved with the sequential DCT.

A much more flexible progressive system is attained at some increase in complexity when successive approximation is added to the spectral selection progression. For a given bit rate, this system typically provides significantly better image quality than spectral selection alone. The net coding efficiency at the completion of the final stage is typically comparable to or slightly better than that achieved with the sequential DCT.

K.7.2 Hierarchical progression

Hierarchical progression permits a sequence of outputs of increasing spatial resolution, and also allows refinement of image quality at a given spatial resolution. Both DCT and spatial versions of the hierarchical progression are allowed, and progressive coding of DCT coefficients may be used in a frame of the DCT hierarchical progression.

The DCT hierarchical progression is intended for applications which need very good compression for a given level of visual distortion; the spatial hierarchical progression is intended for applications which need a simple progression with a truly lossless final stage. Figure K.6 illustrates examples of these two basic hierarchical progressions.

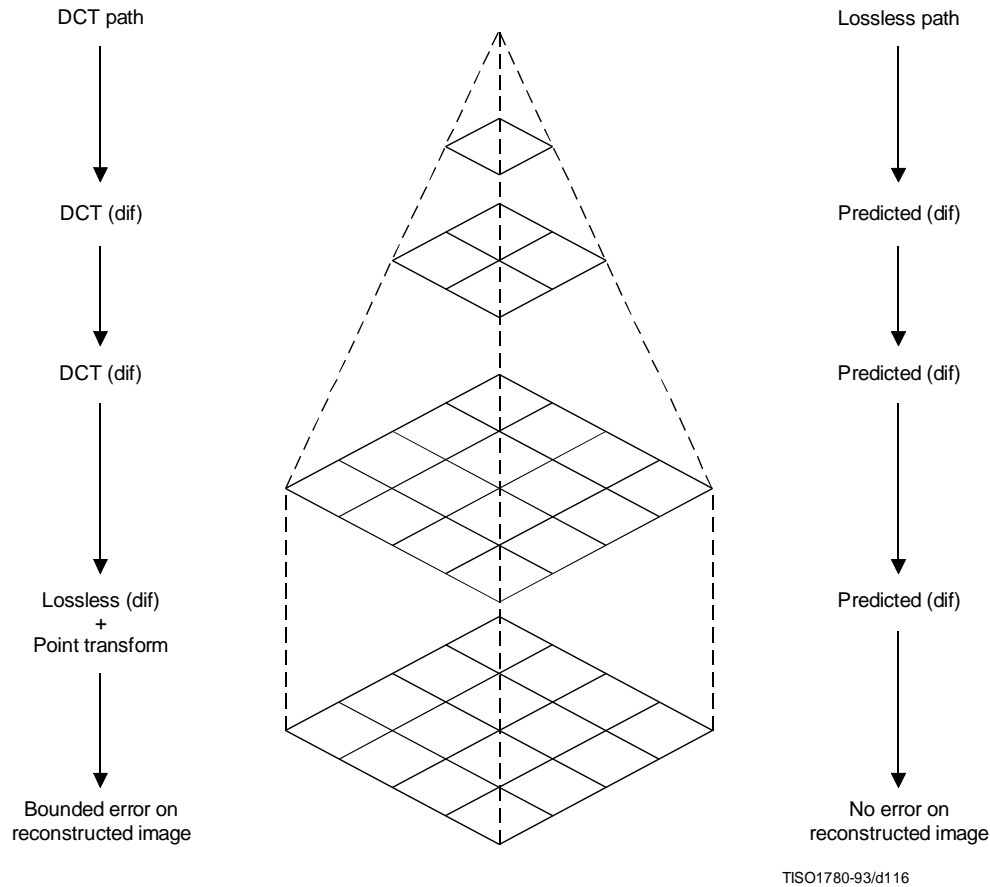


Figure K.6 – Sketch of the basic operations of the hierarchical mode

K.7.2.1 DCT Hierarchical progression

If a DCT hierarchical progression uses reduced spatial resolution, the early stages of the progression can have better image quality for a given bit rate than the early stages of non-hierarchical progressive coding of the DCT coefficients. However, at the point where the distortion between source and output becomes indistinguishable, the coding efficiency achieved with a DCT hierarchical progression is typically significantly lower than the coding efficiency achieved with a non-hierarchical progressive coding of the DCT coefficients.

While the hierarchical DCT progression is intended for lossy progressive coding, a final spatial differential coding stage can be used. When this final stage is used, the output can be almost lossless, limited only by the difference between the encoder and decoder IDCT implementations. Since IDCT implementations can differ significantly, truly lossless coding after a DCT hierarchical progression cannot be guaranteed. An important alternative, therefore, is to use the input point transform of the final lossless differential coding stage to reduce the precision of the differential input. This allows a bounding of the difference between source and output at a significantly lower cost in coded bits than coding of the full precision spatial difference would require.

K.7.2.2 Spatial hierarchical progression

If lossless progression is required, a very simple hierarchical progression may be used in which the spatial lossless coder with point transformed input is used as a first stage. This first stage is followed by one or more spatial differential coding stages. The first stage should be nearly lossless, such that the low order bits which are truncated by the point transform are essentially random – otherwise the compression efficiency will be degraded relative to non-progressive lossless coding.

K.8 Suppression of block-to-block discontinuities in decoded images

A simple technique is available for suppressing the block-to-block discontinuities which can occur in images compressed by DCT techniques.

The first few (five in this example) low frequency DCT coefficients are predicted from the nine DC values of the block and the eight nearest-neighbour blocks, and the predicted values are used to suppress blocking artifacts in smooth areas of the image.

The prediction equations for the first five AC coefficients in the zig-zag sequence are obtained as follows:

K.8.1 AC prediction

The sample field in a 3 by 3 array of blocks (each block containing an 8 × 8 array of samples) is modeled by a two-dimensional second degree polynomial of the form:

$$P(x,y) = A1(x^2y^2) + A2(x^2y) + A3(xy^2) + A4(x^2) + A5(xy) + A6(y^2) + A7(x) + A8(y) + A9$$

The nine coefficients A1 through A9 are uniquely determined by imposing the constraint that the mean of P(x,y) over each of the nine blocks must yield the correct DC-values.

Applying the DCT to the quadratic field predicting the samples in the central block gives a prediction of the low frequency AC coefficients depicted in Figure K.7.

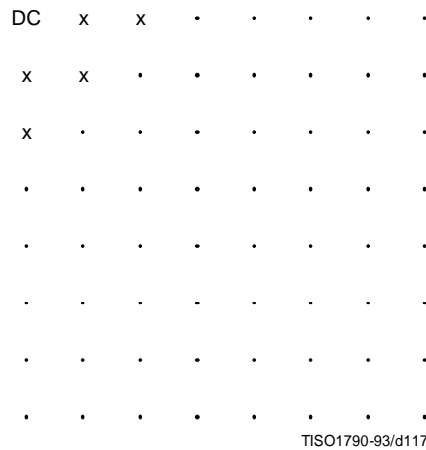


Figure K.7 – DCT array positions of predicted AC coefficients

The prediction equations derived in this manner are as follows:

For the two dimensional array of DC values shown

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.

$$\begin{aligned} QAC_{01} &= ((R_d \times Q_{01}) + (36 \times Q_{00} \times (QDC_4 - QDC_6)))/(256 \times Q_{01}) \\ QAC_{10} &= ((R_d \times Q_{10}) + (36 \times Q_{00} \times (QDC_2 - QDC_8)))/(256 \times Q_{10}) \\ QAC_{20} &= ((R_d \times Q_{20}) + (9 \times Q_{00} \times (QDC_2 + QDC_8 - 2 \times QDC_5)))/(256 \times Q_{20}) \\ QAC_{11} &= ((R_d \times Q_{11}) + (5 \times Q_{00} \times ((QDC_1 - QDC_3) - (QDC_7 - QDC_9)))/(256 \times Q_{11}) \\ QAC_{02} &= ((R_d \times Q_{02}) + (9 \times Q_{00} \times (QDC_4 + QDC_6 - 2 \times QDC_5)))/(256 \times Q_{02}) \end{aligned}$$

where QDC_x and QAC_{xy} are the quantized and scaled DC and AC coefficient values. The constant R_d is added to get a correct rounding in the division. R_d 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, A_1 , 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:

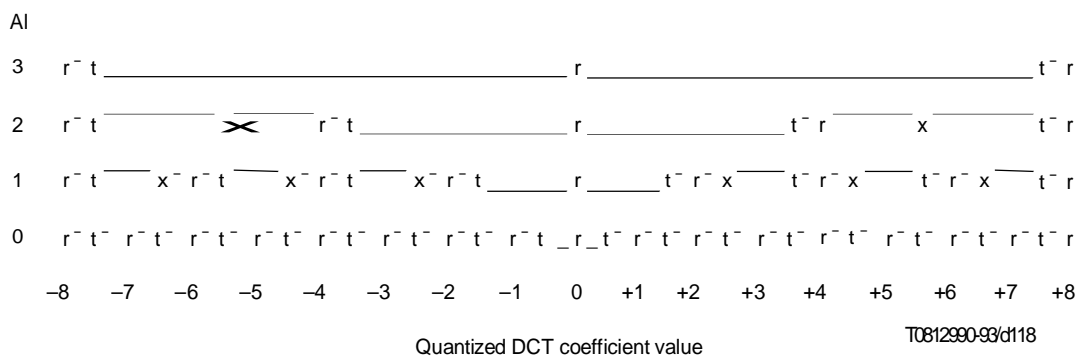


Figure K.8 – Illustration of two reconstruction strategies

The column to the left labelled “ A_1 ” gives the bit position specified in the scan header. The quantized DCT coefficient magnitudes are therefore divided by 2^{A_1} during that scan.

Referring to the final scan ($A_l = 0$), 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 A_l 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 P_t and divide by 2^{P_t} 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 $P_t = 1$:

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 Varembe
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.

ISO/IEC 10918-1 : 1993(E)

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 IECEC*, 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

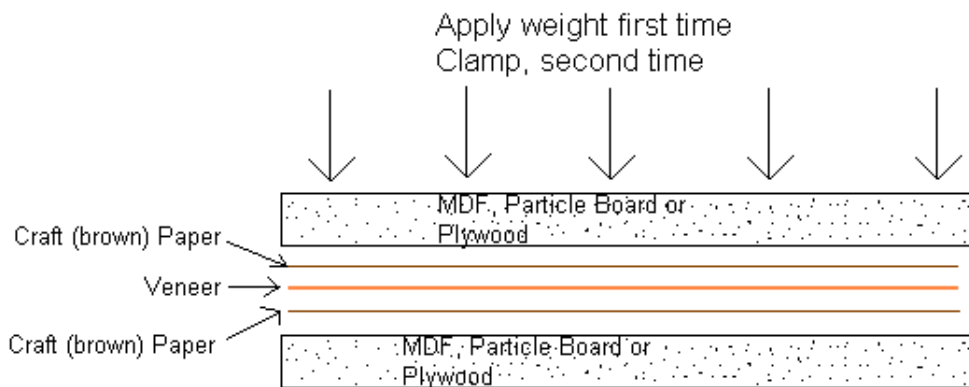
HUFFMAN (D.A.): A Method for the Construction of Minimum Redundancy codes, *Proc. IRE*, Vol. 40, pp. 1098-1101, 1952.

Flattening Wavy Veneer

The grain in wood is particularly important in its effect on the figure of wood. Different types of irregular (highly figured) woods may raise, crack or blister. Some of the best examples of this are the burls and crotch veneers.

The first prerequisite for a good veneering job is that the veneer must be flat, clean and dry. Because of the modern method of manufacturing veneer and the nature of some highly figured woods like burls and crotch having no grain direction, they are most often supplied wavy or buckled. The following procedure should be used to flatten wavy veneer prior to bonding it to a surface.

Prepare a solution of 10 percent glycerine (which can be purchased at a drug store) to 90 percent water. Pour this solution into a clean spray bottle (the type used to spray plants). Set the spray nozzle to spray a very fine mist. Next mist both sides of the veneer. Place the dampened veneer between two sheets of brown craft paper, (purchased at any art supply store) make sure there is no print on the paper. Next, place the veneer and paper between two panels. Any stable, flat panel material like MDF, particleboard or plywood will do. Add a minimum amount of weight to the top and leave overnight. Examine the next day, if the veneer is still not flat, repeat the process using new craft paper, this time the veneer should be flat enough so you can clamp it between the panels. Once again leave overnight.



Once the veneer is flat, do not apply any more solution, but change the craft paper every day for about 5 days allowing the dry craft paper to absorb the moisture that you have added during the flattening process. Keep the veneer clamped between the panels at all times until you are ready to bond it. Make sure you have everything ready to go (glue, clamps, roller etc.) before you remove the veneer from the clamped panels.

[Return To Sal Marino's Homepage](#)

Kepler's laws

Conservation of angular momentum

16 **Kp²**

Kepler

Conservation of angular momentum.

Torque τ is defined as the instantaneous time rate of change of angular momentum l :

$$\tau == dl/dt.$$

equation 1

Angular momentum is a quantity which plays the same part in rotational mechanics as linear momentum does in linear mechanics.

We know from the previous section that $\tau = \mathbf{o}$ -- the Sun never applies a torque to a planet. Therefore dl/dt must also be the zero vector:

$$dl/dt = \mathbf{o}.$$

equation 2

If the time derivative of something is zero, that means that thing does not change as time passes; in other words, it remains constant. This is usually only applied to scalars, however. In vectors, if the time derivative of a vector is the zero vector, then that vector does not change magnitude *or* direction. In other words, the angular momentum vector of a planet is a constant vector:

$$l = \text{constant}.$$

equation 3

Because the Sun does not apply a torque to a planet from its gravitational influence, the angular momentum of the planet remains constant; it is conserved. This is the main concept behind Kepler's second law.

What is the mathematical expression for angular momentum, though? We can find an expression for angular momentum from our expression for torque, substituting in dl/dt for τ :

$$dl/dt = \mathbf{r} \text{ cross } \mathbf{F}.$$

equation 4

We can use Newton's law of motion, $\mathbf{F} = m \mathbf{a}$, and substitute this into our equation:

$$dl/dt = \mathbf{r} \text{ cross } (m \mathbf{a}).$$

equation 5

The acceleration of a body is equal to its instantaneous rate of change of velocity; that is,

$$\mathbf{a} == d\mathbf{v}/dt.$$

equation 6

Making this substitution (and also exploiting the fact that the cross product is associative with respect to scalar factors), we find that

$$dl/dt = \mathbf{r} \text{ cross } (m d\mathbf{v}/dt)$$

equation 7

$$dl/dt = m (\mathbf{r} \text{ cross } d\mathbf{v}/dt).$$

equation 8

If we solve this differential equation, we find that

$$l = m (\mathbf{r} \text{ cross } \mathbf{v}).$$

equation 9

The magnitude of the angular momentum is

$$l = |l|$$

equation 10

$$l = |m (\mathbf{r} \text{ cross } \mathbf{v})|$$

equation 11

$$l = m |\mathbf{r} \text{ cross } \mathbf{v}|.$$

equation 12

This relates the angular momentum of a planet to its mass, position, and velocity.

Navigation.

Erik Max Francis -- TOP

Welcome to my homepage.

⁰e

Physics -- UP

Physics-related information.

⁶Ph

Kepler's laws -- UP

A proof of Kepler's laws.

¹⁶Kp

Kepler's laws: Torque -- PREVIOUS

¹⁶Kp¹

Kepler's laws: Kepler's second law -- NEXT

¹⁶Kp³

Quick links.

Contents of Erik Max Francis' homepages -- CONTENTS

Everything in my homepages.

¹In¹

Feedback -- FEEDBACK

How to send feedback on these pages to the author.

¹In⁵

About Erik Max Francis -- PERSONAL

Information about me.

¹In⁷

Copyright -- COPYRIGHT

Copyright information regarding these pages.

¹In⁴

Copyright © 2003 Erik Max Francis. All rights reserved.

¹⁶Kp²

Web presence provided by
Alcyone Systems

Last updated
2003 Jul 05 14:31

Web design by
7 sisters productions

Kepler

The Alcyone Systems Web Ring (13 sites) >>>

[Alcyone Systems | **Erik Max Francis** | Blackgirl International |
Bosskey.net | Alcyone Systems' CatCam | Crank dot Net |
Hard Science | Losers dot Org | Max Pandaemonium |
Polly Wanna Cracka? | Realpolitik | sade deluxe |
7 Sisters Productions]

Hard Science

Hard science, the easy way.

Max Pandaemonium

Max Pandaemonium's
original music.

Kepler's laws

Kepler's second law

Kepler's second law.

We now proceed to directly address Kepler's second law, the one which states that a ray from the Sun to a planet sweeps out equal areas in equal times. This ray is simply the vector \mathbf{r} that we've been using. (And we shall continue to use it; \mathbf{r} , remember, is defined as the vector from the Sun to the planet.)

What we're looking for is the area that this vector sweeps out. Imagine the planet at some time $t = 0$, and then imagine at a short time afterward $t = \text{deltat}$. In that time, the vector has moved by a short displacement

$$\mathbf{deltar} = \mathbf{r}(t = \text{deltat}) - \mathbf{r}(t = 0). \quad \text{equation 1}$$

The three vectors $\mathbf{r}(t = 0)$, \mathbf{deltar} , and $\mathbf{r}(t = \text{deltat})$ form a triangle. The area of this triangle closely approximates the area swept out by the vector \mathbf{r} during that short time deltat .

We can write this small area represented by this triangle, $\text{delta}A$, as one-half of the parallelogram defined by the vectors \mathbf{r} and \mathbf{deltar} , or

$$\text{delta}A = (1/2) |\mathbf{r} \text{ cross } \mathbf{deltar}|. \quad \text{equation 2}$$

We'll divide both sides of this equation by deltat , the short time involved. Because of this, and the associative properties of the cross product, we find:

$$\text{delta}A/\text{deltat} = (1/2) (1/\text{deltat}) |\mathbf{r} \text{ cross } \mathbf{deltar}| \quad \text{equation 3}$$

$$\text{delta}A/\text{deltat} = (1/2) |(\mathbf{r} \text{ cross } \mathbf{deltar})/\text{deltat}| \quad \text{equation 4}$$

$$\text{delta}A/\text{deltat} = (1/2) |\mathbf{r} \text{ cross } (\mathbf{deltar}/\text{deltat})|. \quad \text{equation 5}$$

As we choose smaller and smaller values of deltat , we get better and better approximations of the area swept out by the ray. If we let deltat approach zero by taking the limit of both sides, the approximation approaches the real value and we find that

$$dA/dt = (1/2) |\mathbf{r} \text{ cross } d\mathbf{r}/dt| \quad \text{equation 6}$$

or

$$dA/dt = (1/2) |\mathbf{r} \text{ cross } \mathbf{v}|. \quad \text{equation 7}$$

Knowing

$$l = m |\mathbf{r} \text{ cross } \mathbf{v}| \quad \text{equation 8}$$

and dividing both sides by m , we find

$$l/m = |\mathbf{r} \text{ cross } \mathbf{v}|. \quad \text{equation 9}$$

We can substitute this into our expression for dA/dt and find that

$$dA/dt = l/(2 m). \quad \text{equation 10}$$

That is, the instantaneous time rate of change of area is the magnitude of the angular momentum divided by twice the mass of the planet. But we know that the mass of the planet is constant, and we also know from our work earlier that the angular momentum vector is constant (and thus its magnitude certainly is). Therefore, the time derivative of area swept out by this ray is constant. In other words, no matter where on the orbit the planet is, its ray still sweeps out the same amount of area. This is Kepler's second law.

Navigation.

Erik Max Francis -- TOP

Welcome to my homepage.

0^e

Physics -- UP

Physics-related information.

6^{Ph}

Kepler's laws -- UP

A proof of Kepler's laws.

16^{Kp}

Kepler's laws: Conservation of angular momentum -- PREVIOUS

16^{Kp^2}

Kepler's laws: Polar basis vectors -- NEXT

16^{Kp^4}

Quick links.

Contents of Erik Max Francis' homepages -- CONTENTS

Everything in my homepages.

1^{In^1}

Feedback -- FEEDBACK

How to send feedback on these pages to the author.

1^{In^5}

About Erik Max Francis -- PERSONAL

Information about me.

1^{In^7}

Copyright -- COPYRIGHT

Copyright information regarding these pages.

1^{In^4}

Copyright © 2003 Erik Max Francis. All rights reserved.

16^{Kp^3}

Web presence provided by
Alcyone Systems

Last updated
2003 Jul 05 14:31

Web design by
7 sisters productions

Kepler

The Alcyone Systems Web Ring (13 sites) >>>

[Alcyone Systems | **Erik Max Francis** | Blackgirl International |
Bosskey.net | Alcyone Systems' CatCam | Crank dot Net |
Hard Science | Losers dot Org | Max Pandaemonium |
Polly Wanna Cracka? | Realpolitik | sade deluxe |
7 Sisters Productions]

Hard Science

Hard science, the easy way.

Max Pandaemonium

Max Pandaemonium's
original music.

Kepler's laws

Polar basis vectors

Polar basis vectors.

Shortly we shall move on to Kepler's first law, which concerns itself with the shape of the orbit that a planet makes around the Sun. Before we do that, we must deal with polar basis vectors.

The polar coordinate system is an effective way of representing the positions of bodies with the angle they make with the origin, and the distance they are away from it. Polar coordinates are useful for dealing with motion *around* a central point -- just the case we have with planets moving around the Sun.

However, to continue with our use of vectors, we must define a few polar basis vectors. Our first defined vector will be the unit radial vector. It will be represented by \mathbf{r} and will be defined as

$$\mathbf{r} == \cos \theta \mathbf{i} + \sin \theta \mathbf{j}.$$

equation 1

Note that this vector is a function of θ ; in other words, the unit vector representing the direction in which the body is located from the Sun is naturally dependent on the angle.

This is a fortunate definition; according to it,

$$\mathbf{r} = r \mathbf{r}$$

equation 2

something we were already using! Therefore we need make no change of notation. Our definition of \mathbf{r} as a polar basis vector merely quantifies our work in the plane of the orbit.

Since we have two Cartesian basis vectors, \mathbf{i} and \mathbf{j} , we should also have two polar basis vectors. The second basis vector, which we shall call the unit transverse vector and represent with $\boldsymbol{\theta}$, is defined as the rate of change of \mathbf{r} with respect to θ :

$$\boldsymbol{\theta} == d\mathbf{r}/d\theta$$

equation 3

$$\boldsymbol{\theta} = -\sin \theta \mathbf{i} + \cos \theta \mathbf{j}.$$

equation 4

This definition means that $\boldsymbol{\theta}$ always points orthogonally to the unit radial vector. This makes it easy to talk about the component of a vector along \mathbf{r} , the radial direction, and the component along $\boldsymbol{\theta}$, the transverse direction.

Note that if we again take the derivative of $\boldsymbol{\theta}$ with respect to θ we find that

$$d\boldsymbol{\theta}/d\theta = -\cos \theta \mathbf{i} - \sin \theta \mathbf{j}$$

equation 5

$$d\boldsymbol{\theta}/d\theta = -(\cos \theta \mathbf{i} + \sin \theta \mathbf{j})$$

equation 6

$$d\boldsymbol{\theta}/d\theta = -\mathbf{r}.$$

equation 7

We shall make use of this later.

Let us, for the sake of an example, see what our velocity vector \mathbf{v} and our angular momentum vector \mathbf{l} would look like in terms of this new polar system. (We shall require them later in the proof anyway.)

Velocity is the instantaneous rate of change of the position of the planet:

$$\mathbf{v} = d\mathbf{r}/dt \tag{equation 8}$$

$$\mathbf{v} = (d/dt) (r \mathbf{r}) \tag{equation 9}$$

$$\mathbf{v} = dr/dt \mathbf{r} + r d\mathbf{r}/dt. \tag{equation 10}$$

But this looks like something we've already dealt with! You may be tempted immediately to substitute \mathbf{theta} in for $d\mathbf{r}/dt$, but remember the definition: $\mathbf{theta} = d\mathbf{r}/dtheta$, something considerably different. We use the chain rule to expand $d\mathbf{r}/dt$ into a form which includes $d\mathbf{r}/dtheta$:

$$\mathbf{v} = dr/dt \mathbf{r} + r dtheta/dt d\mathbf{r}/dtheta. \tag{equation 11}$$

Since $d\mathbf{r}/dtheta$ is \mathbf{theta} , the unit transverse vector, and the angular speed, ω , is defined as

$$\omega == dtheta/dt \tag{equation 12}$$

we can obtain our final expression for velocity in polar coordinates as

$$\mathbf{v} = dr/dt \mathbf{r} + r \omega d\mathbf{r}/dtheta. \tag{equation 13}$$

To find a similar expression for the angular momentum vector \mathbf{l} in polar coordinates, we go back to the expression we found for angular momentum:

$$\mathbf{l} = m (\mathbf{r} \text{ cross } \mathbf{v}). \tag{equation 14}$$

We can substitute $r \mathbf{r}$ for \mathbf{r} and the expression we just found for \mathbf{v} (I told you we'd need it), and get

$$\mathbf{l} = m [(r \mathbf{r}) \text{ cross } (dr/dt \mathbf{r} + r \omega d\mathbf{r}/dtheta)]. \tag{equation 15}$$

We expand this expression to obtain

$$\mathbf{l} = m [(r \mathbf{r}) \text{ cross } (dr/dt \mathbf{r} + r \omega d\mathbf{r}/dtheta)] \tag{equation 16}$$

$$\mathbf{l} = m (r \mathbf{r}) \text{ cross } (dr/dt \mathbf{r}) + m (r \mathbf{r}) \text{ cross } (r \omega d\mathbf{r}/dtheta) \tag{equation 17}$$

$$\mathbf{l} = m r dr/dt (\mathbf{r} \text{ cross } \mathbf{r}) + m r^2 \omega (\mathbf{r} \text{ cross } d\mathbf{r}/dtheta). \tag{equation 18}$$

Since, again, a vector crossed with itself is the zero vector, the first term evaluates to zero and we find that

$$\mathbf{l} = m r^2 \omega (\mathbf{r} \text{ cross } d\mathbf{r}/dtheta). \tag{equation 19}$$

Since $\mathbf{r} \text{ cross } d\mathbf{r}/dtheta = \mathbf{k}$, our final expression for the angular momentum vector is

$$\mathbf{l} = m r^2 \omega \mathbf{k}. \tag{equation 20}$$

As we took the magnitude of this vector before, we shall do it again:

$$l = |\mathbf{l}| \tag{equation 21}$$

$$l = |m r^2 \omega \mathbf{k}| \tag{equation 22}$$

$$l = m r^2 \omega |\mathbf{k}|$$

equation 23

$$l = m r^2 \omega$$

equation 24

as the magnitude of a unit vector is, by definition, unity.

Navigation.

Erik Max Francis -- TOP

Welcome to my homepage.

⁰e

Physics -- UP

Physics-related information.

⁶Ph

Kepler's laws -- UP

A proof of Kepler's laws.

¹⁶Kp

Kepler's laws: Kepler's second law -- PREVIOUS

¹⁶Kp³

Kepler's laws: Kepler's first law -- NEXT

¹⁶Kp⁵

Quick links.

Contents of Erik Max Francis' homepages -- CONTENTS

Everything in my homepages.

¹In¹

Feedback -- FEEDBACK

How to send feedback on these pages to the author.

¹In⁵

About Erik Max Francis -- PERSONAL

Information about me.

¹In⁷

Copyright -- COPYRIGHT

Copyright information regarding these pages.

¹In⁴

Copyright © 2003 Erik Max Francis. All rights reserved.

¹⁶Kp⁴

Web presence provided by
Alcyone Systems

Last updated
2003 Jul 05 14:31

Web design by
7 sisters productions

Kepler

The Alcyone Systems Web Ring (13 sites) >>>

[Alcyone Systems | **Erik Max Francis** | Blackgirl International |
Bosskey.net | Alcyone Systems' CatCam | Crank dot Net |
Hard Science | Losers dot Org | Max Pandaemonium |
Polly Wanna Cracka? | Realpolitik | sade deluxe |
7 Sisters Productions]

Hard Science

Hard science, the easy way.

Max Pandaemonium

Max Pandaemonium's
original music.

Kepler's laws

Kepler's first law

16 **Kp**⁵

Kepler

Kepler's first law.

Now that we have polar basis vectors under our wing (and the polar representations of velocity and angular momentum), we are ready to proceed with the proof of Kepler's first law -- that the orbits of planets are ellipses with the Sun at one focus.

To begin with, we will start off by applying Newton's law of motion and Newton's law of universal gravitation together to find that

$$m \mathbf{a} = (-G m M/r^2) \mathbf{r} \quad \text{equation 1}$$

and, dividing both sides of the equation by m ,

$$\mathbf{a} = (-G M/r^2) \mathbf{r}. \quad \text{equation 2}$$

Recalling our work with polar basis vectors, we know $d\mathbf{theta}/d\theta = -\mathbf{r}$. Solving for \mathbf{r} and applying the chain rule, we find that

$$\mathbf{r} = -dt/d\theta \, d\mathbf{theta}/dt \quad \text{equation 3}$$

$$\mathbf{r} = -1/\omega \, d\mathbf{theta}/dt. \quad \text{equation 4}$$

Substituting this into our equation for \mathbf{a} we find

$$\mathbf{a} = (-G M/r^2) (-1/\omega) \, d\mathbf{theta}/dt \quad \text{equation 5}$$

$$\mathbf{a} = (G M)/(r^2 \omega) \, d\mathbf{theta}/dt. \quad \text{equation 6}$$

If we multiply the right side of the equation by m/m (which is unity), we obtain

$$\mathbf{a} = (G m M)/(m r^2 \omega) \, d\mathbf{theta}/dt. \quad \text{equation 7}$$

But $l = m r^2 \omega$ (I told you this would come in handy as well), so we can rewrite this as

$$\mathbf{a} = (G m M/l) \, d\mathbf{theta}/dt. \quad \text{equation 8}$$

We can multiply both sides by $l/(G m M)$ and find

$$l/(G m M) \mathbf{a} = d\mathbf{theta}/dt. \quad \text{equation 9}$$

But we know that $\mathbf{a} = d\mathbf{v}/dt$, and can substitute accordingly:

$$l/(G m M) \, d\mathbf{v}/dt = d\mathbf{theta}/dt. \quad \text{equation 10}$$

This is a differential equation that we can now solve. Upon solving it, we find that

$$l/(G m M) \mathbf{v} = \mathbf{theta} + \mathbf{C} \quad \text{equation 11}$$

where \mathbf{C} is some constant vector. We'll solve this for \mathbf{v} and find that

$$\mathbf{v} = (G m M/l) (\mathbf{theta} + \mathbf{C}). \quad \text{equation 12}$$

This is a general solution to the differential equation.

But we're not finished. This doesn't tell us much about the shape of a planet's orbit, although all the pieces are there. This is the general solution, and it could be an orbit of *any* of the possible shapes (though we can't be sure what they are yet) or *any* of the possible orientations. We're interested in knowing the shape, of course, so we want to restrict the possible orientations.

To do that, we'll take a special case. It makes sense to have perihelion -- that is, closest approach to the Sun -- at time $t = 0$. We'll restrict the orientation so that, when perihelion occurs, the planet lies along the zero radian line from the Sun (or, in Cartesian terminology, along the positive x -axis) -- that is, $\theta = 0$. At this point, \mathbf{r} , the position vector of the planet, will have only a component in the positive x -axis. We'll also assume that the planet orbits the Sun counterclockwise, through increasing measures of angles. If this is the case, then the velocity \mathbf{v} at the instant of perihelion should be orthogonal to the position vector \mathbf{r} , and it should have only a component in the positive y -axis.

According to our expression for \mathbf{v} , we have a scalar times the vector quantity $\mathbf{\hat{\theta}} + \mathbf{C}$.

$$\mathbf{\hat{\theta}}(\theta = 0) = \mathbf{j};$$

equation 13

that is, the unit transverse vector points "up" when the unit radial vector points "right." Since, at $t = 0$, $\mathbf{\hat{\theta}}$ points entirely in the y -direction, then our constant vector \mathbf{C} must only have a component in the y -axis -- this is the only way to get a resultant vector (\mathbf{v}) that points entirely in the y -direction. So, we can rewrite \mathbf{C} as a scalar times the unit basis vector in the y -direction:

$$\mathbf{C} = e \mathbf{j}$$

equation 14

where e is some scalar constant. (You will find that we have a very good reason for choosing the letter e in this case.) Substituting this into our equation for \mathbf{v} , we get

$$\mathbf{v} = (G m M/l) (\mathbf{\hat{\theta}} + e \mathbf{j}).$$

equation 15

This is the specific case when we want the orbit oriented so that perihelion occurs at $t = \theta = 0$.

Now we are ready to finish up the problem. We can dot both sides of the equation with $\mathbf{\hat{\theta}}$ and get:

$$\mathbf{v} \cdot \mathbf{\hat{\theta}} = (G m M/l) (\mathbf{\hat{\theta}} + e \mathbf{j}) \cdot \mathbf{\hat{\theta}}$$

equation 16

$$\mathbf{v} \cdot \mathbf{\hat{\theta}} = (G m M/l) [\mathbf{\hat{\theta}} \cdot \mathbf{\hat{\theta}} + e (\mathbf{j} \cdot \mathbf{\hat{\theta}})].$$

equation 17

A vector dotted with itself yields the square of that vector's magnitude, so $\mathbf{\hat{\theta}} \cdot \mathbf{\hat{\theta}} = 1$. Simplifying $\mathbf{v} \cdot \mathbf{\hat{\theta}}$, we find

$$\mathbf{v} \cdot \mathbf{\hat{\theta}} = (dr/dt \mathbf{r} + r \omega \mathbf{\hat{\theta}}) \cdot \mathbf{\hat{\theta}}$$

equation 18

$$\mathbf{v} \cdot \mathbf{\hat{\theta}} = dr/dt (\mathbf{r} \cdot \mathbf{\hat{\theta}}) + r \omega (\mathbf{\hat{\theta}} \cdot \mathbf{\hat{\theta}}).$$

equation 19

But the dot product of two orthogonal vectors is zero, so $\mathbf{r} \cdot \mathbf{\hat{\theta}} = 0$. We also already know that $\mathbf{\hat{\theta}} \cdot \mathbf{\hat{\theta}} = 1$. Therefore

$$\mathbf{v} \cdot \mathbf{\hat{\theta}} = r \omega.$$

equation 20

The last part of our problem is finding an expression for $\mathbf{j} \cdot \mathbf{\hat{\theta}}$. We know that $\mathbf{\hat{\theta}} = -\sin \theta \mathbf{i} + \cos \theta \mathbf{j}$ (by definition), so

$$\mathbf{j} \cdot \boldsymbol{\theta} = \mathbf{j} \cdot (-\sin \theta \mathbf{i} + \cos \theta \mathbf{j}) \quad \text{equation 21}$$

$$\mathbf{j} \cdot \boldsymbol{\theta} = \cos \theta. \quad \text{equation 22}$$

We have the three pieces of the puzzle, so we'll put them together to find that

$$r \omega = (G m M/l) (1 + e \cos \theta). \quad \text{equation 23}$$

We see $r \omega$ on the left side of this equation. We know that $l = m r^2 \omega$, and it would be nice to get rid of the mess that way. So we'll multiply both sides of the equation by $m r$ to get

$$m r^2 \omega = (G m^2 M/l) [r (1 + e \cos \theta)]. \quad \text{equation 24}$$

Replacing the left side of the equation by l and moving the constants to the left side of our equation, we find that

$$l = (G m^2 M/l) [r (1 + e \cos \theta)]. \quad \text{equation 25}$$

We're almost finished now. Here it is clear that we have an explicit function in terms of r and θ -- in other words, this should be the polar equation for our planet's orbit! All we need do now is solve for r :

$$r = [l^2/(G m^2 M)]/(1 + e \cos \theta). \quad \text{equation 26}$$

The equation of a conic section with focus-directrix distance p and eccentricity e is represented by the polar equation

$$r = e p/(1 + e \cos \theta). \quad \text{equation 27}$$

But this is exactly what we have, given that

$$e p = l^2/(G m^2 M). \quad \text{equation 28}$$

The focus-directrix distance should be a constant, which p is: l , G , m , and M are all individually constant; therefore the expression $l^2/(G m^2 M)$ must also be constant. Therefore, Newton's laws of motion and universal gravitation dictate that the orbits of planets follow conic sections. This is Kepler's first law.

... Well, *almost*. Kepler's first law actually states that planets follow the paths of *ellipses*. An ellipse is only one type of conic section. So why is an ellipse allowed while the others are not?

Because the others *are* allowed -- we just don't call them *planets*. When Kepler said *planet*, he meant a body which returns to our skies over and over again. This means that the curve representing the orbit must be closed -- that is, it must continue to retrace itself over and over. The only two conic sections which are closed are the circle and the ellipse (the circle is just a special case of the ellipse anyway). Thus, any body in the skies we see over and over must be orbiting the Sun in an ellipse.

The other two conic sections -- the parabola and hyperbola -- are open curves and correspond to a position where the body has sufficient velocity to escape from the Sun's gravity well. The body would approach the Sun from an infinite distance, round the Sun rapidly, and then recede away into the infinite abyss, never to be seen again.

So we have proved an extension of Kepler's first law: A body influenced by the Sun's

gravity follows a path defined by a conic section with the Sun at one focus.

Navigation.

Erik Max Francis -- TOP

Welcome to my homepage.

⁰e

Physics -- UP

Physics-related information.

⁶Ph

Kepler's laws -- UP

A proof of Kepler's laws.

¹⁶Kp

Kepler's laws: Polar basis vectors -- PREVIOUS

¹⁶Kp⁴

Kepler's laws: Kepler's third law -- NEXT

¹⁶Kp⁶

Quick links.

Contents of Erik Max Francis' homepages -- CONTENTS

Everything in my homepages.

¹In¹

Feedback -- FEEDBACK

How to send feedback on these pages to the author.

¹In⁵

About Erik Max Francis -- PERSONAL

Information about me.

¹In⁷

Copyright -- COPYRIGHT

Copyright information regarding these pages.

¹In⁴

Copyright © 2003 Erik Max Francis. All rights reserved.

¹⁶Kp⁵

Web presence provided by
Alcyone Systems

Last updated
2003 Jul 05 14:31

Web design by
7 sisters productions

Kepler

The Alcyone Systems Web Ring (13 sites) >>>>

[Alcyone Systems | **Erik Max Francis** | Blackgirl International |
Bosskey.net | Alcyone Systems' CatCam | Crank dot Net |
Hard Science | Losers dot Org | Max Pandaemonium |
Polly Wanna Cracka? | Realpolitik | sade deluxe |
7 Sisters Productions]

Hard Science

Hard science, the easy way.

Max Pandaemonium

Max Pandaemonium's
original music.

Kepler's laws

Kepler's third law

Kepler's third law.

After getting Kepler's first and second laws (though not in that order) out of our way, we're ready to tackle Kepler's third and final law.

Actually, after all of the trouble we've gone through, the third law is easy to prove and seems almost an afterthought. The third law relates the period of a planet's orbit, T , to the length of its semimajor axis, a . It states that the square of the period of the orbit (T^2) is proportional to the cube of the semimajor axis (a^3), and further that the constant of proportionality is independent of the individual planets; in other words, each and every planet has the same constant of proportionality.

Note that since we're talking about a *planet's* period, we clearly must be referring to a planet which orbits the Sun in a closed curve -- that is, either a circle or an ellipse: a conic section with $e < 1$.

We'll start with the expression we derived for the rate of change of the area that the Sun-planet ray is sweeping out (Kepler's second law):

$$dA/dt = l/(2 m). \tag{equation 1}$$

We'll multiply both sides by dt and find

$$dA = [l/(2 m)] dt. \tag{equation 2}$$

We can integrate once around the orbit (from 0 to A and from 0 to T) to get an expression relating the total area of the orbit to the period of the orbit:

$$A = [l/(2 m)] T. \tag{equation 3}$$

If we square both sides and solve for T^2 , we find that

$$T^2 = 4 (m^2/l^2) A^2. \tag{equation 4}$$

The area A of an ellipse is $\pi a b$, where a is the length of the semimajor axis and b the length of the semiminor axis. Thus our expression of T^2 becomes

$$T^2 = 4 \pi^2 (m^2/l^2) a^2 b^2. \tag{equation 5}$$

We know that b is related to a and c , the focus-center distance, by $a^2 = b^2 + c^2$, so $b^2 = a^2 - c^2$:

$$T^2 = 4 \pi^2 (m^2/l^2) a^2 (a^2 - c^2). \tag{equation 6}$$

Since $c = a e$, we find that

$$T^2 = 4 \pi^2 (m^2/l^2) a^2 (a^2 - a^2 e^2) \tag{equation 7}$$

$$T^2 = 4 \pi^2 (m^2/l^2) a^2 [a^2 (1 - e^2)] \tag{equation 8}$$

$$T^2 = 4 \pi^2 (m^2/l^2) a^4 (1 - e^2). \tag{equation 9}$$

Since we're dealing here with ellipses (and circles), we can use a property of ellipse geometry which indicates that

$$e p = a (1 - e^2). \tag{equation 10}$$

This relates the semimajor axis a and the eccentricity e of an ellipse to its focus-directrix distance p . If we factor out $a (1 - e^2)$ from our expression of T^2 and replace it with $e p$ we find that

$$T^2 = 4 \pi^2 (m^2/l^2) a^3 [a (1 - e^2)] \tag{equation 11}$$

$$T^2 = 4 \pi^2 (m^2/l^2) a^3 (e p). \tag{equation 12}$$

Here we have an expression which indicates that T^2 is proportional to a^3 ; but the constant of proportionality doesn't look like it is the same quantity for all planets -- after all, it seems to be a function of m and l , which are certainly different for each planet. Thus, we'll continue on with our proof. We know from Kepler's first law that $e p = l^2/(G m^2 M)$. We can use this information to substitute into our expression for T^2 to find that

$$T^2 = 4 \pi^2 (m^2/l^2) l^2/(G m^2 M) a^3. \tag{equation 13}$$

The m^2 and l^2 cancel, and we are left with

$$T^2 = (4 \pi^2)/(G M) a^3. \tag{equation 14}$$

The constant of proportionality, $(4 \pi^2)/(G M)$, is the same quantity for all planets -- it depends only on G , the constant of universal gravitation, and M , the mass of the Sun. Therefore, the square of the period of a planet is proportional to the cube of the length of the semimajor axis, and this proportionality is the same for all planets. This is Kepler's third law.

Quod erat demonstrandum.

Navigation.

Erik Max Francis -- TOP

Welcome to my homepage.

₀e

Physics -- UP

Physics-related information.

₆Ph

Kepler's laws -- UP

A proof of Kepler's laws.

₁₆Kp

Kepler's laws: Kepler's first law -- PREVIOUS

₁₆Kp⁵

Kepler's laws: Commentary -- NEXT

₁₆Kp⁷

Quick links.

Contents of Erik Max Francis' homepages -- CONTENTS

Everything in my homepages.

₁In¹

Feedback -- FEEDBACK

How to send feedback on these pages to the author.

1In⁵

About Erik Max Francis -- PERSONAL

Information about me.

1In⁷

Copyright -- COPYRIGHT

Copyright information regarding these pages.

1In⁴

Copyright © 2003 Erik Max Francis. All rights reserved.

16Kp⁶

Web presence provided by
Alcyone Systems

Last updated
2003 Oct 18 02:06

Web design by
7 sisters productions

Kepler

The Alcyone Systems Web Ring (13 sites) >>>

[Alcyone Systems | **Erik Max Francis** | Blackgirl International |
Bosskey.net | Alcyone Systems' CatCam | Crank dot Net |
Hard Science | Losers dot Org | Max Pandaemonium |
Polly Wanna Cracka? | Realpolitik | sade deluxe |
7 Sisters Productions]

Hard Science

Hard science, the easy way.

Max Pandaemonium

Max Pandaemonium's
original music.

Special Numbers

Measurements & Conversions

Physics and Other Science

Mathreference.net

Arithmetic

Algebra

Geometry

Trig

calculus

Statistics

Algebra



"Give me where to stand, and I will move
the earth."

Archimedes
287 BC - 212 BC

Page created and maintained by **Jennifer Ledbetter**. 2003 all rights reserved.

..... Tools used for this site Links About the author

Special Numbers

Measurements & Conversions

Physics and Other Science

Mathreference.net

Arithmetic

Algebra

Geometry

Trig

calculus

Statistics

Arithmetic



Fibonacci
1170 - 1250

Page created and maintained by **Jennifer Ledbetter**. 2003 all rights reserved.

..... Tools used for this site Links About the author

Mathreference.net



Geometry

- Angles in the Four Quadrants
- Polygons
- Areas
- Euclid's Elements
- Logic - Truth Table
- Pythagorean Triples

"The laws of nature are but the mathematical thoughts of God."

Euclid of Alexandria
~325 BC - ~265 BC

Page created and maintained by **Jennifer Ledbetter**. 2003 all rights reserved.

..... Tools used for this site Links About the author

Special Numbers

Measurements & Conversions

Physics and Other Science

Mathreference.net

Arithmetic

Algebra

Geometry

Trig

calculus

Statistics

Statistics



"One must know oneself, if this does not serve to discover truth, it at least serves as a rule of life and there is nothing better."

Blaise Pascal
1623 - 1662

Page created and maintained by **Jennifer Ledbetter**. 2003 all rights reserved.

..... [Tools used for this site](#) [Links](#) [About the author](#)

Special Numbers

Measurements & Conversions

Physics and Other Science

Mathreference.net

Arithmetic

Algebra

Geometry

Trig

calculus

Statistics

Trig



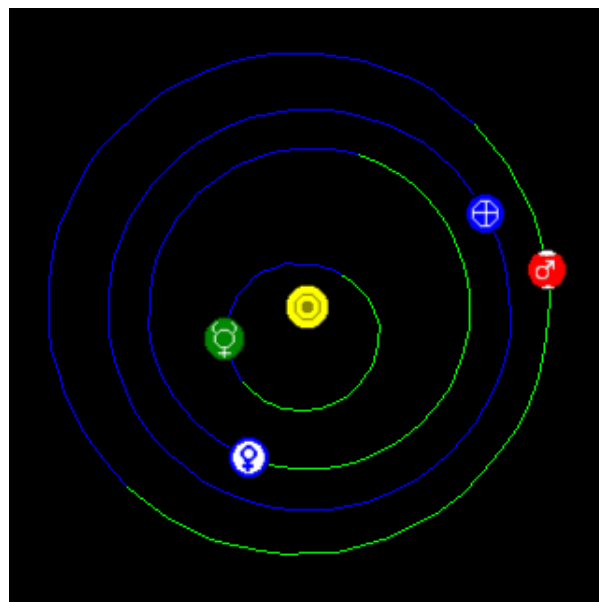
"Rest satisfied with doing well, and leave others to talk of you as they please."

Pythagoras of Samos
~569 BC - ~475 BC

Page created and maintained by **Jennifer Ledbetter**. 2003 all rights reserved.

..... Tools used for this site Links About the author

Solar System Live



Time: Now UTC: 2003-10-21 19:49:29 Julian: 2452934.32603
 Update **Show:** Icons Images
Display: Full system Inner system **Size:** 320 **Stereo:** Cross Wall
Orbits: Real Logarithmic Equal
Observing site: Lat. 47° N S Long. 7° E W
Heliocentric: Lat. 90° N S Long. 0°

Ephemeris:

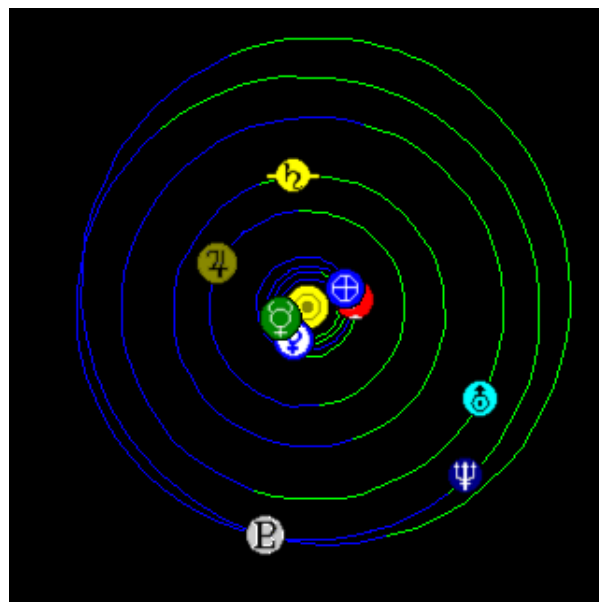
	Right Ascension	Declination	Distance (AU)	From 47°N 7°E:		
				Altitude	Azimuth	
Sun	13h 43m 58s	-10° 45.6'	0.995	-33.441	112.343	Set
Mercury	13h 36m 11s	-8° 56.9'	1.408	-33.253	115.493	Set
Venus	14h 50m 17s	-16° 7.4'	1.617	-26.406	94.465	Set
Moon	10h 45m 43s	+13° 16.8'	59.5 ER	-29.374	171.937	Set
Mars	22h 27m 37s	-12° 52.4'	0.562	30.078	-3.030	Up
Jupiter	10h 52m 32s	+8° 10.7'	6.022	-34.255	169.301	Set
Saturn	6h 57m 9s	+22° 3.2'	8.726	-7.603	-134.312	Set
Uranus	22h 5m 44s	-12° 32.4'	19.503	30.408	3.150	Up
Neptune	20h 51m 24s	-17° 38.8'	29.847	22.553	22.079	Up
Pluto	17h 10m 50s	-14° 6.3'	31.333	-1.369	70.621	Set

To track an asteroid or comet, paste orbital elements below: Echo elements

[Return to Solar System Live](#)
[Details](#)
[Credits](#)
[Customise](#)
[Help](#)

by John Walker

Solar System Live



Time: Now UTC: 2003-10-21 19:50:20 Julian: 2452934.32662
 Update **Show:** Icons Images
Display: Full system Inner system **Size:** 320 **Stereo:** Cross Wall
Orbits: Real Logarithmic Equal
Observing site: Lat. 47° N S Long. 7° E W
Heliocentric: Lat. 90° N S Long. 0°

Ephemeris:

	Right Ascension	Declination	Distance (AU)	From 47°N 7°E:		
				Altitude	Azimuth	
Sun	13h 43m 58s	-10° 45.6'	0.995	-33.575	112.535	Set
Mercury	13h 36m 11s	-8° 57.0'	1.408	-33.384	115.689	Set
Venus	14h 50m 18s	-16° 7.4'	1.617	-26.551	94.626	Set
Moon	10h 45m 45s	+13° 16.6'	59.5 ER	-29.396	172.165	Set
Mars	22h 27m 37s	-12° 52.4'	0.562	30.085	-2.791	Up
Jupiter	10h 52m 32s	+8° 10.7'	6.022	-34.281	169.554	Set
Saturn	6h 57m 9s	+22° 3.2'	8.726	-7.499	-134.143	Set
Uranus	22h 5m 44s	-12° 32.4'	19.503	30.400	3.391	Up
Neptune	20h 51m 24s	-17° 38.8'	29.847	22.498	22.290	Up
Pluto	17h 10m 50s	-14° 6.3'	31.333	-1.506	70.775	Set

To track an asteroid or comet, paste orbital elements below: Echo elements

[Return to Solar System Live](#)
[Details](#)
[Credits](#)
[Customise](#)
[Help](#)

by John Walker



[Home](#) | [Physics](#) | [Astronomy](#) | [Courses](#)

[Contents](#) | [Planet Finder](#) | [BinoSky](#) | [Variable star calendar](#) | [Links](#)

Planet Finder

This applet shows the locations of the planets, stars, moon, and sun in the sky from any location and for any date and time. If your browser doesn't run the applet properly, please see my notes on Java problems. If this is your first time using the applet, you might want to read the introductory notes further down on the page.

[[Afrikaanse weergawe](#) | [Dansk version](#) | [Deutsche Version](#) | [Dutch version](#) | [?????? ??????????](#) | [English version](#) | [Version Español](#) | [Version française](#) | [Versione italiana](#) | [Vers?o Portuguesa](#) | [Svensk version](#)]

Introductory Notes

(1) The program needs to know your location in order to predict what the sky will look like. Initially, it just guesses the biggest city in your time zone. If this is incorrect, you can set your location either by typing in your latitude and longitude or by choosing a city from the list (which only includes those with populations greater than 3 million).

- (2) Use 24-hour time for setting the time, e.g. 13:00 means 1 pm.
- (3) In the U.S., daylight savings time lasts from 2 am on the first sunday of April until 2 am on the last sunday of October. The program handles this by default. If you are not in the U.S., you may need to set daylight savings time manually.
- (4) The compass directions may look wrong, but that is because the screen represents the way the sky looks when you look straight up. The compass directions are therefore a mirror image of the compass directions on a map, which represents a view of the land looking down from above.
- (5) The spherical sky has to be projected onto the flat screen. This projection produces distortion, just as a map of the earth inevitably has some distortion. The greatest distortion occurs near the horizon.
- (6) The disks of the sun, moon, and planets are not drawn to scale. Their brightnesses are given as magnitudes to the right of their names. A more negative magnitude means a brighter planet. The magnitudes given for Saturn do not include the brightness of the rings, so Saturn will usually be brighter than indicated.
- (7) This program is only designed to have a limited degree of accuracy, sufficient for most naked-eye astronomy applications. More detailed information is given below.

Including Planet Finder in Your Own Page

You are welcome either to link to my Planet Finder page or to have the applet appear in your own page. A possible advantage of the latter is that in your html code you can set an appropriate language, latitude, and longitude for the people who visit your page. The following html example shows how you would include the Danish-language version ("da") in your page, with the default latitude and longitude set for Copenhagen:

```
<applet archive="http://www.lightandmatter.com/PlanetFinder.jar"
  code="PlanetFinder.class"
  width=600 height=400
  alt="Sorry, your browser does not support Java.">
<param name="language" value="da">
<param name="latitude" value="56">
<param name="longitude" value="12">
</applet>
```

Note that, due to a bug, the latitude and longitude as set in the html code must be integers. (The user can type in noninteger values in the applet, and that works fine.)

Translating the Program into Your Language

If you would like to do this, please e-mail me your translation of the following words and phrases. First comes a list of phrases to translate that occur on the web page. After that is some computer code, in which you only need to translate what's inside the quotes. You can just indicate accents in any way that's convenient for you (e.g., "e'" and "o'", or with Unicode), and I'll try to code them correctly for the computer. Of course you should change the references to Italian so they refer to your own language. Thanks!!

```
Planet Finder [can be translated as something like "Planets Now"
  or "Planets Today" -- I leave it up to your judgment to pick whatever
  best in your language]
"Light and Matter" web page [You can decide whether quotes are appropriate in your l
```

an applet that shows the current locations of the stars and planets in the night sky
Italian version of Planet Finder
Sorry, but I am unable to respond to e-mail in Italian.
Light and Matter home page (in English)
Thanks to (your name) for the Italian translation.

```
daylightSavings = "Daylight savings";
auto = "Auto";
manual = "Manual";
northLetter = "N";
southLetter = "S";
eastLetter = "E";
westLetter = "W";
sky = "Sky";
innerSolarSystem = "Inner solar system";
outerSolarSystem = "Outer solar system";
update = "Update";
nakedEyePlanetsOnly = "Naked-eye planets only";
timeZone = "Time zone";
    monthNames[0] = "Jan"; monthNames[3] = "Apr"; monthNames[6] = "Jul"; mont
    monthNames[1] = "Feb"; monthNames[4] = "May"; monthNames[7] = "Aug"; mont
    monthNames[2] = "Mar"; monthNames[5] = "Jun"; monthNames[8] = "Sep"; mont
planetNames[0] = "Mercury"; planetNames[3] = "Mars"; planetNames[
planetNames[1] = "Venus"; planetNames[4] = "Jupiter"; planetNames[7] = "Ne
planetNames[2] = "Earth"; planetNames[5] = "Saturn"; planetNames[8] = "Pl
sun = "Sun";
moon = "Moon";
latitude = "Latitude";
longitude = "Longitude";
localTime = "Local time";
```

Known Bugs

- (1) The applet does not quit gracefully, so when you reload the page you get a bunch of Java errors.
- (2) If you program the default latitude and longitude into your html, you must use integers.

Open Source Stuff

The source code is subject to the GPL open source license. Source code: [here](#).

Technical Notes on Precision

Planet Finder calculates everything based almost entirely on Kepler's laws, so it is no good for extremely high-precision work or for projections many centuries into the past or future. The only nonkeplerian behavior incorporated into the program is the precession of the moon's orbit and the first few correction terms for the moon's motion given by Paul Schlyter. The parallax due to the rotation of the earth is taken into account, but the program does not calculate the effect of the earth's oblateness or of the time taken for light to travel from the planets to us. The sun's motion relative to the solar system's center of mass is taken into account. Refraction in the earth's atmosphere is not taken into account, which is why the times of sunrise and sunset will not agree exactly with what is listed in the newspaper. For high-precision calculations, you may want to check out Solar System Live, a web page that displays

a map of the locations of the planets in the solar system (not a map of the sky), or Your Sky, which is a planetarium applet like this one but fancier and not quite as easy to use. .

Acknowledgments

I am very grateful to Paul Schlyter and Dave Williams for their help with the celestial mechanics. Thanks to Kristian Pedersen for the Danish translation.

[[Top of page](#) | [Home](#) | [Site Map](#) | [Privacy](#) | [Contact](#)]

(c) Copyright 1998 Benjamin Crowell. All rights reserved.
Hubble Space Telescope deep field courtesy of STScI.

PNG (Portable Network Graphics) Specification

Version 1.0

For list of authors, see Credits (Chapter 19).

Status of this document

This document has been reviewed by W3C members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

A list of current W3C Recommendations and other technical documents can be found at <http://www.w3.org/pub/WWW/TR/>.

The Consortium staff have encouraged the development of PNG, as have Compuserve, Inc. Most of the work has been done by the PNG Development Group, png-group@w3.org. The Consortium does not currently have plans to work on any future versions of PNG, though were the necessity to arise, and were an activity in that area to receive the support of Members, the Consortium could in principle support some future activity.

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`.

Contents

1	Introduction	5
2	Data Representation	6
2.1	Integers and byte order	6
2.2	Color values	6
2.3	Image layout	7
2.4	Alpha channel	8
2.5	Filtering	8
2.6	Interlaced data order	9
2.7	Gamma correction	10
2.8	Text strings	10
3	File Structure	10
3.1	PNG file signature	10
3.2	Chunk layout	11
3.3	Chunk naming conventions	11
3.4	CRC algorithm	13
4	Chunk Specifications	14
4.1	Critical chunks	14
4.1.1	IHDR Image header	14
4.1.2	PLTE Palette	15
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	18
4.2.4	hIST Image histogram	19
4.2.5	pHYs Physical pixel dimensions	19
4.2.6	sBIT Significant bits	20
4.2.7	tEXt Textual data	20
4.2.8	tIME Image last-modification time	22
4.2.9	tRNS Transparency	22
4.2.10	zTXt Compressed textual data	23
4.3	Summary of standard chunks	24
4.4	Additional chunk types	25
5	Deflate/Inflate Compression	25

6	Filter Algorithms	26
6.1	Filter types	26
6.2	Filter type 0: None	27
6.3	Filter type 1: Sub	27
6.4	Filter type 2: Up	28
6.5	Filter type 3: Average	29
6.6	Filter type 4: Paeth	29
7	Chunk Ordering Rules	30
7.1	Behavior of PNG editors	31
7.2	Ordering of ancillary chunks	31
7.3	Ordering of critical chunks	32
8	Miscellaneous Topics	32
8.1	File name extension	32
8.2	Internet media type	32
8.3	Macintosh file layout	32
8.4	Multiple-image extension	33
8.5	Security considerations	33
9	Recommendations for Encoders	33
9.1	Sample depth scaling	34
9.2	Encoder gamma handling	35
9.3	Encoder color handling	37
9.4	Alpha channel creation	38
9.5	Suggested palettes	38
9.6	Filter selection	39
9.7	Text chunk processing	39
9.8	Use of private chunks	40
9.9	Private type and method codes	41
10	Recommendations for Decoders	41
10.1	Error checking	41
10.2	Pixel dimensions	42
10.3	Truecolor image handling	42
10.4	Sample depth rescaling	42
10.5	Decoder gamma handling	43
10.6	Decoder color handling	45
10.7	Background color	46
10.8	Alpha channel processing	46
10.9	Progressive display	50
10.10	Suggested-palette and histogram usage	51
10.11	Text chunk processing	52

11 Glossary	52
12 Appendix: Rationale	56
12.1 Why a new file format?	56
12.2 Why these features?	56
12.3 Why not these features?	57
12.4 Why not use format X?	58
12.5 Byte order	58
12.6 Interlacing	59
12.7 Why gamma?	59
12.8 Non-premultiplied alpha	60
12.9 Filtering	60
12.10 Text strings	61
12.11 PNG file signature	61
12.12 Chunk layout	62
12.13 Chunk naming conventions	62
12.14 Palette histograms	63
13 Appendix: Gamma Tutorial	64
14 Appendix: Color Tutorial	70
15 Appendix: Sample CRC Code	74
16 Appendix: Online Resources	76
17 Appendix: Revision History	76
18 References	77
19 Credits	79

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 non-rectangular 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 must 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.

In this specification, the word “must” indicates a mandatory requirement, while “should” indicates recommended behavior.

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 must 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 sample depth. Note that the maximum value at a given sample depth is $(2^{\text{sampledepth}})-1$, not $2^{\text{sampledepth}}$.

Sample values are not necessarily linear; the γ AMA 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 have to cope with. See Recommendations for Encoders: Sample depth scaling (Section 9.1) and Recommendations for Decoders: Sample 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 $(2^{\text{bitdepth}})-1$ 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. Encoders and decoders must 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 must 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) in files conforming to this version of PNG.

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 or per palette index (not per pixel). 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.

The sample depth is the same as the bit depth except in the case of color type 3, in which the sample depth is always 8 bits.

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 must not appear for color types 0 and 4. If this chunk does appear, it must precede the first IDAT chunk. There must not 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 will need to 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:

1. 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.
2. 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.)
3. 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 .. (2bitdepth)-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 .. (2bitdepth)-1
```

```
Green: 2 bytes, range 0 .. (2bitdepth)-1
```

```
Blue: 2 bytes, range 0 .. (2bitdepth)-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 pixel size or aspect ratio 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 depths can be used, and further specifies that sample values should be scaled to the full range of possible values at the sample 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 sample depth not directly supported by PNG. We recommend that an encoder emit an `sBIT` chunk if it has converted the data from a lower sample 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: Sample depth scaling (Section 9.1) and Recommendations for Decoders: Sample 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 must 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 $(2^{\text{bitdepth}})-1$).

For color type 2 (truecolor), the `tRNS` chunk contains a single RGB color value, stored in the format:

```
Red:    2 bytes, range 0 .. (2bitdepth)-1
Green:  2 bytes, range 0 .. (2bitdepth)-1
Blue:   2 bytes, range 0 .. (2bitdepth)-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 $(2^{\text{bitdepth}})-1$).

`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` or at `png-group@w3.org`.

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/>](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.

6.1 Filter types

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 will have to be stored by the decoder at all times. Even though some filter types do not refer to the prior scanline, the decoder will always need to 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).

6.2 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.

6.3 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:

$$\text{Sub}(x) = \text{Raw}(x) - \text{Raw}(x - \text{bpp})$$

where x ranges from zero to the number of bytes representing the scanline minus one, $\text{Raw}(x)$ 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 $\text{Raw}(x) = 0$.

To reverse the effect of the Sub filter after decompression, output the following value:

$$\text{Sub}(x) + \text{Raw}(x - \text{bpp})$$

(computed mod 256), where Raw refers to the bytes already decoded.

6.4 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:

$$\text{Up}(x) = \text{Raw}(x) - \text{Prior}(x)$$

where x ranges from zero to the number of bytes representing the scanline minus one, $\text{Raw}(x)$ refers to the raw data byte at that byte position in the scanline, and $\text{Prior}(x)$ 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 $\text{Prior}(x) = 0$ for all x .

To reverse the effect of the Up filter after decompression, output the following value:

$$\text{Up}(x) + \text{Prior}(x)$$

(computed mod 256), where Prior refers to the decoded bytes of the prior scanline.

6.5 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:

$$\text{Average}(x) = \text{Raw}(x) - \text{floor}((\text{Raw}(x-\text{bpp}) + \text{Prior}(x)) / 2)$$

where x ranges from zero to the number of bytes representing the scanline minus one, $\text{Raw}(x)$ refers to the raw data byte at that byte position in the scanline, $\text{Prior}(x)$ 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 $\text{Raw}(x-\text{bpp}) + \text{Prior}(x)$ must be formed without overflow (using at least nine-bit arithmetic). $\text{floor}()$ 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 $\text{Raw}(x) = 0$. On the first scanline of an image (or of a pass of an interlaced image), assume $\text{Prior}(x) = 0$ for all x .

To reverse the effect of the Average filter after decompression, output the following value:

$$\text{Average}(x) + \text{floor}((\text{Raw}(x-\text{bpp}) + \text{Prior}(x)) / 2)$$

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.

6.6 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:

$$\text{Paeth}(x) = \text{Raw}(x) - \text{PaethPredictor}(\text{Raw}(x-\text{bpp}), \text{Prior}(x), \text{Prior}(x-\text{bpp}))$$

where x ranges from zero to the number of bytes representing the scanline minus one, $\text{Raw}(x)$ refers to the raw data byte at that byte position in the scanline, $\text{Prior}(x)$ 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 $\text{Raw}(x) = 0$ and $\text{Prior}(x) = 0$. On the first scanline of an image (or of a pass of an interlaced image), assume $\text{Prior}(x) = 0$ for all x .

To reverse the effect of the Paeth filter after decompression, output the following value:

$$\text{Paeth}(x) + \text{PaethPredictor}(\text{Raw}(x-\text{bpp}), \text{Prior}(x), \text{Prior}(x-\text{bpp}))$$

(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 must 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 Sample depth scaling

When encoding input samples that have a sample depth that cannot be directly represented in PNG, the encoder must scale the samples up to a sample depth that is allowed by PNG. The most accurate scaling method is the linear equation

$$\text{output} = \text{ROUND}(\text{input} * \text{MAXOUTSAMPLE} / \text{MAXINSAMPLE})$$

where the input samples range from 0 to MAXINSAMPLE and the outputs range from 0 to MAXOUTSAMPLE (which is $(2^{\text{sampledepth}})-1$).

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 sample depths, some encoders may choose to use it. Zero-fill must **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 sample 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 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

$$\text{sample} = \text{ROUND}((\text{intensity} \wedge \text{encoder_gamma}) * \text{MAXSAMPLE})$$

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

$$\text{screen_gc} = \text{viewing_gamma} / \text{display_gamma}$$

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 re-quantization 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`:

$$\text{file_gamma} = \text{viewing_gamma} / \text{display_gamma}$$

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 has to 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 γ AMA 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.

Image authors 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. Note that color images with 256 or fewer colors should almost always be stored in indexed color format; truecolor format is likely to be much larger.

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 should be output using uncompressed `tEXt` chunks. In particular, it is recommended that the basic title and author keywords should 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. Numbers greater than 127 must be used 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 should 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 Sample depth rescaling

Decoders may wish to scale PNG data to a lesser sample depth (data 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

$$\text{output} = \text{ROUND}(\text{input} * \text{MAXOUTSAMPLE} / \text{MAXINSAMPLE})$$

where

```
MAXINSAMPLE = (2^sampledepth)-1
MAXOUTSAMPLE = (2^desired_sampledepth)-1
```

A slightly less accurate conversion is achieved by simply shifting right by `sampledepth-desired_sampledepth` 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 sample 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 should 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 = insample / MAXINSAMPLE
bright = gbright ^ (1.0 / file_gamma)
vbright = bright ^ viewing_gamma
gcvideo = vbright ^ (1.0 / display_gamma)
fbval = ROUND(gcvideo * MAXFBVAL)
```

where `MAXINSAMPLE` 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), `insample` 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 will need to 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) should 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

$$\text{output} = \text{alpha} * \text{foreground} + (1-\text{alpha}) * \text{background}$$

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 sample 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_sample_depth) - 1;
11 bg_maxsample = (1 < bg_sample_depth) - 1;
12 fb_maxsample = (1 < frame_buffer_sample_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:

1. 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(compix, 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 will need to be replaced by copies of lines 17-23, but processing background instead of foreground pixel values.

2. If the sample depths 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.
3. When the sample 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.
4. 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 need to 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.

5. 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.
6. 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

a^b

Exponentiation; a raised to the power b . 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 must give a zero result.

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.

Bit depth

The number of bits per palette index (in indexed-color PNGs) or per sample (in other color types). This is the same value that appears in IHDR.

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.

Chromaticity

A pair of values x,y that precisely specify the hue, though not the absolute brightness, of a perceived color.

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.

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.

Critical chunk

A chunk that must be understood and processed by the decoder in order to produce a meaningful image from a PNG file.

CRT

Cathode Ray Tube: a common type of computer display hardware.

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

$$\text{output} = \text{input} ^ \text{gamma}$$

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. When discussing physical data layout (in particular, in Image layout, Section 2.3), we use “sample” to mean a number stored in the image array. It would be more precise but much less readable to say “sample or palette index” in that context. Elsewhere in the specification, “sample” means a color value or alpha value. In the indexed-color case, these are palette entries not palette indexes.

Sample depth

The precision, in bits, of color values and alpha values. In indexed-color PNGs the sample depth is always 8 by definition of the `PLTE` chunk. In other color types it is the same as the bit depth.

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.

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, PNG software must support full deflate/inflate; any software that does not is not 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 ought to 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.

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. 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. (But see Multiple-image extension, Section 8.4.)

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, bitwise 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 should 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 needs to 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

$$\text{output} = \text{input} \wedge \text{gamma}$$

where \wedge 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 needs to 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

$$\text{file_gamma} = \text{camera_gamma} * \text{encoding_gamma}$$

display_gamma

the gamma of the “display system” downstream of the frame buffer. This is

$$\text{display_gamma} = \text{LUT_gamma} * \text{CRT_gamma}$$

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

$$\text{file_gamma} * \text{decoding_gamma} * \text{display_gamma} = \text{viewing_gamma}$$

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

$$\text{display_gamma} = 2.5 / \text{SGI_system_gamma}$$

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

$$\begin{aligned} V_{out} &= 4.5 * V_{in} && \text{if } V_{in} < 0.018 \\ V_{out} &= 1.099 * (V_{in}^{0.45}) - 0.099 && \text{if } V_{in} \geq 0.018 \end{aligned}$$

where V_{in} and V_{out} 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 γ AMA 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 γ AMA 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 c HRM chunk is used, together with the γ AMA 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 :

$$m = \begin{matrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{matrix}$$

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:

$$\begin{matrix} X \\ Y \\ Z \end{matrix} = m \begin{matrix} R \\ G \\ B \end{matrix}$$

In other words, $X = X_r * R + X_g * G + X_b * B$, and similarly for Y and Z. You can go the other way too:

$$\begin{matrix} R \\ G \\ B \end{matrix} = im \begin{matrix} X \\ Y \\ Z \end{matrix}$$

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, \wedge 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 or at png-group@w3.org.

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.

On 1 October 1996, the PNG specification was approved as a W3C (World Wide Web Consortium) Recommendation.

At that time, it was awaiting publication as an Informational RFC.

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 sample 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
- Clarified terminology, notably sample depth vs. bit depth
- 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/%7Epoynnton/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.berkeley.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, Aleks.Jakulin@snet.fri.uni-lj.si
- Neal Kettler, kettler@cs.colostate.edu
- Tom Lane, tgl@sss.pgh.pa.us
- Alexander Lehmann, alex@hal.rhein-main.de
- Chris Lilley, chris@w3.org
- Dave Martindale, davem@cs.ubc.ca
- Owen Mortensen, 104707.650@compuserve.com
- Keith S. Pickens, ksp@swri.edu
- Robert P. Poole, lionboy@primenet.com
- Glenn Randers-Pehrson, glennrp@arl.mil or randeg@alumni.rpi.edu
- Greg Roelofs, newt@pobox.com
- Willem van Schaik, willem@gintic.gov.sg
- Guy Schalnat
- Paul Schmidt, pschmidt@photodex.com
- Tim Wegner, 71320.675@compuserve.com
- Jeremy Wohl, jeremyw@anders.com

The authors wish to acknowledge the contributions of the Portable Network Graphics mailing list, the readers of `comp.graphics`, and the members of the World Wide Web Consortium (W3C).

The Adam7 interlacing scheme is not patented and it is not the intention of the originator of that scheme to patent it. The scheme may be freely used by all PNG implementations. The name “Adam7” may be freely used to describe interlace method 1 of the PNG specification.

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.

COPYRIGHT NOTICE

Copyright © 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.

End of PNG Specification