IJG JPEG LIBRARY: SYSTEM ARCHITECTURE

Copyright (C) 1991-2013, Thomas G. Lane, Guido Vollbeding.

This file is part of the Independent JPEG Group's software.

For conditions of distribution and use, see the accompanying README file.

This file provides an overview of the architecture of the IJG JPEG software;

that is, the functions of the various modules in the system and the interfaces

between modules. For more precise details about any data structure or calling

convention, see the include files and comments in the source code.

We assume that the reader is already somewhat familiar with the JPEG standard.

The README file includes references for learning about JPEG. The file

libjpeg.txt describes the library from the viewpoint of an application

programmer using the library; it's best to read that file before this one.

Also, the file coderules.txt describes the coding style conventions we use.

In this document, JPEG-specific terminology follows the JPEG standard:

A "component" means a color channel, e.g., Red or Luminance.

A "sample" is a single component value (i.e., one number in the image data).

A "coefficient" is a frequency coefficient (a DCT transform output number).

A "block" is an array of samples or coefficients.

An "MCU" (minimum coded unit) is an interleaved set of blocks of size

determined by the sampling factors, or a single block in a

noninterleaved scan.

We do not use the terms "pixel" and "sample" interchangeably. When we say

pixel, we mean an element of the full-size image, while a sample is an element

of the downsampled image. Thus the number of samples may vary across

components while the number of pixels does not. (This terminology is not used

rigorously throughout the code, but it is used in places where confusion would

otherwise result.)

\*\*\* System features \*\*\*

The IJG distribution contains two parts:

\* A subroutine library for JPEG compression and decompression.

\* cjpeg/djpeg, two sample applications that use the library to transform

JFIF JPEG files to and from several other image formats.

cjpeg/djpeg are of no great intellectual complexity: they merely add a simple

command-line user interface and I/O routines for several uncompressed image

formats. This document concentrates on the library itself.

We desire the library to be capable of supporting all JPEG baseline, extended

sequential, and progressive DCT processes. The library does not support the

hierarchical or lossless processes defined in the standard.

Within these limits, any set of compression parameters allowed by the JPEG

spec should be readable for decompression. (We can be more restrictive about

what formats we can generate.) Although the system design allows for all

parameter values, some uncommon settings are not yet implemented and may

never be; nonintegral sampling ratios are the prime example. Furthermore,

we treat 8-bit vs. 12-bit data precision as a compile-time switch, not a

run-time option, because most machines can store 8-bit pixels much more

compactly than 12-bit.

By itself, the library handles only interchange JPEG datastreams --- in

particular the widely used JFIF file format. The library can be used by

surrounding code to process interchange or abbreviated JPEG datastreams that

are embedded in more complex file formats. (For example, libtiff uses this

library to implement JPEG compression within the TIFF file format.)

The library includes a substantial amount of code that is not covered by the

JPEG standard but is necessary for typical applications of JPEG. These

functions preprocess the image before JPEG compression or postprocess it after

decompression. They include colorspace conversion, downsampling/upsampling,

and color quantization. This code can be omitted if not needed.

A wide range of quality vs. speed tradeoffs are possible in JPEG processing,

and even more so in decompression postprocessing. The decompression library

provides multiple implementations that cover most of the useful tradeoffs,

ranging from very-high-quality down to fast-preview operation. On the

compression side we have generally not provided low-quality choices, since

compression is normally less time-critical. It should be understood that the

low-quality modes may not meet the JPEG standard's accuracy requirements;

nonetheless, they are useful for viewers.

\*\*\* Portability issues \*\*\*

Portability is an essential requirement for the library. The key portability

issues that show up at the level of system architecture are:

1. Memory usage. We want the code to be able to run on PC-class machines

with limited memory. Images should therefore be processed sequentially (in

strips), to avoid holding the whole image in memory at once. Where a

full-image buffer is necessary, we should be able to use either virtual memory

or temporary files.

2. Near/far pointer distinction. To run efficiently on 80x86 machines, the

code should distinguish "small" objects (kept in near data space) from

"large" ones (kept in far data space). This is an annoying restriction, but

fortunately it does not impact code quality for less brain-damaged machines,

and the source code clutter turns out to be minimal with sufficient use of

pointer typedefs.

3. Data precision. We assume that "char" is at least 8 bits, "short" and

"int" at least 16, "long" at least 32. The code will work fine with larger

data sizes, although memory may be used inefficiently in some cases. However,

the JPEG compressed datastream must ultimately appear on external storage as a

sequence of 8-bit bytes if it is to conform to the standard. This may pose a

problem on machines where char is wider than 8 bits. The library represents

compressed data as an array of values of typedef JOCTET. If no data type

exactly 8 bits wide is available, custom data source and data destination

modules must be written to unpack and pack the chosen JOCTET datatype into

8-bit external representation.

\*\*\* System overview \*\*\*

The compressor and decompressor are each divided into two main sections:

the JPEG compressor or decompressor proper, and the preprocessing or

postprocessing functions. The interface between these two sections is the

image data that the official JPEG spec regards as its input or output: this

data is in the colorspace to be used for compression, and it is downsampled

to the sampling factors to be used. The preprocessing and postprocessing

steps are responsible for converting a normal image representation to or from

this form. (Those few applications that want to deal with YCbCr downsampled

data can skip the preprocessing or postprocessing step.)

Looking more closely, the compressor library contains the following main

elements:

Preprocessing:

\* Color space conversion (e.g., RGB to YCbCr).

\* Edge expansion and downsampling. Optionally, this step can do simple

smoothing --- this is often helpful for low-quality source data.

JPEG proper:

\* MCU assembly, DCT, quantization.

\* Entropy coding (sequential or progressive, Huffman or arithmetic).

In addition to these modules we need overall control, marker generation,

and support code (memory management & error handling). There is also a

module responsible for physically writing the output data --- typically

this is just an interface to fwrite(), but some applications may need to

do something else with the data.

The decompressor library contains the following main elements:

JPEG proper:

\* Entropy decoding (sequential or progressive, Huffman or arithmetic).

\* Dequantization, inverse DCT, MCU disassembly.

Postprocessing:

\* Upsampling. Optionally, this step may be able to do more general

rescaling of the image.

\* Color space conversion (e.g., YCbCr to RGB). This step may also

provide gamma adjustment [ currently it does not ].

\* Optional color quantization (e.g., reduction to 256 colors).

\* Optional color precision reduction (e.g., 24-bit to 15-bit color).

[This feature is not currently implemented.]

We also need overall control, marker parsing, and a data source module.

The support code (memory management & error handling) can be shared with

the compression half of the library.

There may be several implementations of each of these elements, particularly

in the decompressor, where a wide range of speed/quality tradeoffs is very

useful. It must be understood that some of the best speedups involve

merging adjacent steps in the pipeline. For example, upsampling, color space

conversion, and color quantization might all be done at once when using a

low-quality ordered-dither technique. The system architecture is designed to

allow such merging where appropriate.

Note: it is convenient to regard edge expansion (padding to block boundaries)

as a preprocessing/postprocessing function, even though the JPEG spec includes

it in compression/decompression. We do this because downsampling/upsampling

can be simplified a little if they work on padded data: it's not necessary to

have special cases at the right and bottom edges. Therefore the interface

buffer is always an integral number of blocks wide and high, and we expect

compression preprocessing to pad the source data properly. Padding will occur

only to the next block (block\_size-sample) boundary. In an interleaved-scan

situation, additional dummy blocks may be used to fill out MCUs, but the MCU

assembly and disassembly logic will create or discard these blocks internally.

(This is advantageous for speed reasons, since we avoid DCTing the dummy

blocks. It also permits a small reduction in file size, because the

compressor can choose dummy block contents so as to minimize their size

in compressed form. Finally, it makes the interface buffer specification

independent of whether the file is actually interleaved or not.)

Applications that wish to deal directly with the downsampled data must

provide similar buffering and padding for odd-sized images.

\*\*\* Poor man's object-oriented programming \*\*\*

It should be clear by now that we have a lot of quasi-independent processing

steps, many of which have several possible behaviors. To avoid cluttering the

code with lots of switch statements, we use a simple form of object-style

programming to separate out the different possibilities.

For example, two different color quantization algorithms could be implemented

as two separate modules that present the same external interface; at runtime,

the calling code will access the proper module indirectly through an "object".

We can get the limited features we need while staying within portable C.

The basic tool is a function pointer. An "object" is just a struct

containing one or more function pointer fields, each of which corresponds to

a method name in real object-oriented languages. During initialization we

fill in the function pointers with references to whichever module we have

determined we need to use in this run. Then invocation of the module is done

by indirecting through a function pointer; on most machines this is no more

expensive than a switch statement, which would be the only other way of

making the required run-time choice. The really significant benefit, of

course, is keeping the source code clean and well structured.

We can also arrange to have private storage that varies between different

implementations of the same kind of object. We do this by making all the

module-specific object structs be separately allocated entities, which will

be accessed via pointers in the master compression or decompression struct.

The "public" fields or methods for a given kind of object are specified by

a commonly known struct. But a module's initialization code can allocate

a larger struct that contains the common struct as its first member, plus

additional private fields. With appropriate pointer casting, the module's

internal functions can access these private fields. (For a simple example,

see jdatadst.c, which implements the external interface specified by struct

jpeg\_destination\_mgr, but adds extra fields.)

(Of course this would all be a lot easier if we were using C++, but we are

not yet prepared to assume that everyone has a C++ compiler.)

An important benefit of this scheme is that it is easy to provide multiple

versions of any method, each tuned to a particular case. While a lot of

precalculation might be done to select an optimal implementation of a method,

the cost per invocation is constant. For example, the upsampling step might

have a "generic" method, plus one or more "hardwired" methods for the most

popular sampling factors; the hardwired methods would be faster because they'd

use straight-line code instead of for-loops. The cost to determine which

method to use is paid only once, at startup, and the selection criteria are

hidden from the callers of the method.

This plan differs a little bit from usual object-oriented structures, in that

only one instance of each object class will exist during execution. The

reason for having the class structure is that on different runs we may create

different instances (choose to execute different modules). You can think of

the term "method" as denoting the common interface presented by a particular

set of interchangeable functions, and "object" as denoting a group of related

methods, or the total shared interface behavior of a group of modules.

\*\*\* Overall control structure \*\*\*

We previously mentioned the need for overall control logic in the compression

and decompression libraries. In IJG implementations prior to v5, overall

control was mostly provided by "pipeline control" modules, which proved to be

large, unwieldy, and hard to understand. To improve the situation, the

control logic has been subdivided into multiple modules. The control modules

consist of:

1. Master control for module selection and initialization. This has two

responsibilities:

1A. Startup initialization at the beginning of image processing.

The individual processing modules to be used in this run are selected

and given initialization calls.

1B. Per-pass control. This determines how many passes will be performed

and calls each active processing module to configure itself

appropriately at the beginning of each pass. End-of-pass processing,

where necessary, is also invoked from the master control module.

Method selection is partially distributed, in that a particular processing

module may contain several possible implementations of a particular method,

which it will select among when given its initialization call. The master

control code need only be concerned with decisions that affect more than

one module.

2. Data buffering control. A separate control module exists for each

inter-processing-step data buffer. This module is responsible for

invoking the processing steps that write or read that data buffer.

Each buffer controller sees the world as follows:

input data => processing step A => buffer => processing step B => output data

| | |

------------------ controller ------------------

The controller knows the dataflow requirements of steps A and B: how much data

they want to accept in one chunk and how much they output in one chunk. Its

function is to manage its buffer and call A and B at the proper times.

A data buffer control module may itself be viewed as a processing step by a

higher-level control module; thus the control modules form a binary tree with

elementary processing steps at the leaves of the tree.

The control modules are objects. A considerable amount of flexibility can

be had by replacing implementations of a control module. For example:

\* Merging of adjacent steps in the pipeline is done by replacing a control

module and its pair of processing-step modules with a single processing-

step module. (Hence the possible merges are determined by the tree of

control modules.)

\* In some processing modes, a given interstep buffer need only be a "strip"

buffer large enough to accommodate the desired data chunk sizes. In other

modes, a full-image buffer is needed and several passes are required.

The control module determines which kind of buffer is used and manipulates

virtual array buffers as needed. One or both processing steps may be

unaware of the multi-pass behavior.

In theory, we might be able to make all of the data buffer controllers

interchangeable and provide just one set of implementations for all. In

practice, each one contains considerable special-case processing for its

particular job. The buffer controller concept should be regarded as an

overall system structuring principle, not as a complete description of the

task performed by any one controller.

\*\*\* Compression object structure \*\*\*

Here is a sketch of the logical structure of the JPEG compression library:

|-- Colorspace conversion

|-- Preprocessing controller --|

| |-- Downsampling

Main controller --|

| |-- Forward DCT, quantize

|-- Coefficient controller --|

|-- Entropy encoding

This sketch also describes the flow of control (subroutine calls) during

typical image data processing. Each of the components shown in the diagram is

an "object" which may have several different implementations available. One

or more source code files contain the actual implementation(s) of each object.

The objects shown above are:

\* Main controller: buffer controller for the subsampled-data buffer, which

holds the preprocessed input data. This controller invokes preprocessing to

fill the subsampled-data buffer, and JPEG compression to empty it. There is

usually no need for a full-image buffer here; a strip buffer is adequate.

\* Preprocessing controller: buffer controller for the downsampling input data

buffer, which lies between colorspace conversion and downsampling. Note

that a unified conversion/downsampling module would probably replace this

controller entirely.

\* Colorspace conversion: converts application image data into the desired

JPEG color space; also changes the data from pixel-interleaved layout to

separate component planes. Processes one pixel row at a time.

\* Downsampling: performs reduction of chroma components as required.

Optionally may perform pixel-level smoothing as well. Processes a "row

group" at a time, where a row group is defined as Vmax pixel rows of each

component before downsampling, and Vk sample rows afterwards (remember Vk

differs across components). Some downsampling or smoothing algorithms may

require context rows above and below the current row group; the

preprocessing controller is responsible for supplying these rows via proper

buffering. The downsampler is responsible for edge expansion at the right

edge (i.e., extending each sample row to a multiple of block\_size samples);

but the preprocessing controller is responsible for vertical edge expansion

(i.e., duplicating the bottom sample row as needed to make a multiple of

block\_size rows).

\* Coefficient controller: buffer controller for the DCT-coefficient data.

This controller handles MCU assembly, including insertion of dummy DCT

blocks when needed at the right or bottom edge. When performing

Huffman-code optimization or emitting a multiscan JPEG file, this

controller is responsible for buffering the full image. The equivalent of

one fully interleaved MCU row of subsampled data is processed per call,

even when the JPEG file is noninterleaved.

\* Forward DCT and quantization: Perform DCT, quantize, and emit coefficients.

Works on one or more DCT blocks at a time. (Note: the coefficients are now

emitted in normal array order, which the entropy encoder is expected to

convert to zigzag order as necessary. Prior versions of the IJG code did

the conversion to zigzag order within the quantization step.)

\* Entropy encoding: Perform Huffman or arithmetic entropy coding and emit the

coded data to the data destination module. Works on one MCU per call.

For progressive JPEG, the same DCT blocks are fed to the entropy coder

during each pass, and the coder must emit the appropriate subset of

coefficients.

In addition to the above objects, the compression library includes these

objects:

\* Master control: determines the number of passes required, controls overall

and per-pass initialization of the other modules.

\* Marker writing: generates JPEG markers (except for RSTn, which is emitted

by the entropy encoder when needed).

\* Data destination manager: writes the output JPEG datastream to its final

destination (e.g., a file). The destination manager supplied with the

library knows how to write to a stdio stream or to a memory buffer;

for other behaviors, the surrounding application may provide its own

destination manager.

\* Memory manager: allocates and releases memory, controls virtual arrays

(with backing store management, where required).

\* Error handler: performs formatting and output of error and trace messages;

determines handling of nonfatal errors. The surrounding application may

override some or all of this object's methods to change error handling.

\* Progress monitor: supports output of "percent-done" progress reports.

This object represents an optional callback to the surrounding application:

if wanted, it must be supplied by the application.

The error handler, destination manager, and progress monitor objects are

defined as separate objects in order to simplify application-specific

customization of the JPEG library. A surrounding application may override

individual methods or supply its own all-new implementation of one of these

objects. The object interfaces for these objects are therefore treated as

part of the application interface of the library, whereas the other objects

are internal to the library.

The error handler and memory manager are shared by JPEG compression and

decompression; the progress monitor, if used, may be shared as well.

\*\*\* Decompression object structure \*\*\*

Here is a sketch of the logical structure of the JPEG decompression library:

|-- Entropy decoding

|-- Coefficient controller --|

| |-- Dequantize, Inverse DCT

Main controller --|

| |-- Upsampling

|-- Postprocessing controller --| |-- Colorspace conversion

|-- Color quantization

|-- Color precision reduction

As before, this diagram also represents typical control flow. The objects

shown are:

\* Main controller: buffer controller for the subsampled-data buffer, which

holds the output of JPEG decompression proper. This controller's primary

task is to feed the postprocessing procedure. Some upsampling algorithms

may require context rows above and below the current row group; when this

is true, the main controller is responsible for managing its buffer so as

to make context rows available. In the current design, the main buffer is

always a strip buffer; a full-image buffer is never required.

\* Coefficient controller: buffer controller for the DCT-coefficient data.

This controller handles MCU disassembly, including deletion of any dummy

DCT blocks at the right or bottom edge. When reading a multiscan JPEG

file, this controller is responsible for buffering the full image.

(Buffering DCT coefficients, rather than samples, is necessary to support

progressive JPEG.) The equivalent of one fully interleaved MCU row of

subsampled data is processed per call, even when the source JPEG file is

noninterleaved.

\* Entropy decoding: Read coded data from the data source module and perform

Huffman or arithmetic entropy decoding. Works on one MCU per call.

For progressive JPEG decoding, the coefficient controller supplies the prior

coefficients of each MCU (initially all zeroes), which the entropy decoder

modifies in each scan.

\* Dequantization and inverse DCT: like it says. Note that the coefficients

buffered by the coefficient controller have NOT been dequantized; we

merge dequantization and inverse DCT into a single step for speed reasons.

When scaled-down output is asked for, simplified DCT algorithms may be used

that need fewer coefficients and emit fewer samples per DCT block, not the

full 8x8. Works on one DCT block at a time.

\* Postprocessing controller: buffer controller for the color quantization

input buffer, when quantization is in use. (Without quantization, this

controller just calls the upsampler.) For two-pass quantization, this

controller is responsible for buffering the full-image data.

\* Upsampling: restores chroma components to full size. (May support more

general output rescaling, too. Note that if undersized DCT outputs have

been emitted by the DCT module, this module must adjust so that properly

sized outputs are created.) Works on one row group at a time. This module

also calls the color conversion module, so its top level is effectively a

buffer controller for the upsampling->color conversion buffer. However, in

all but the highest-quality operating modes, upsampling and color

conversion are likely to be merged into a single step.

\* Colorspace conversion: convert from JPEG color space to output color space,

and change data layout from separate component planes to pixel-interleaved.

Works on one pixel row at a time.

\* Color quantization: reduce the data to colormapped form, using either an

externally specified colormap or an internally generated one. This module

is not used for full-color output. Works on one pixel row at a time; may

require two passes to generate a color map. Note that the output will

always be a single component representing colormap indexes. In the current

design, the output values are JSAMPLEs, so an 8-bit compilation cannot

quantize to more than 256 colors. This is unlikely to be a problem in

practice.

\* Color reduction: this module handles color precision reduction, e.g.,

generating 15-bit color (5 bits/primary) from JPEG's 24-bit output.

Not quite clear yet how this should be handled... should we merge it with

colorspace conversion???

Note that some high-speed operating modes might condense the entire

postprocessing sequence to a single module (upsample, color convert, and

quantize in one step).

In addition to the above objects, the decompression library includes these

objects:

\* Master control: determines the number of passes required, controls overall

and per-pass initialization of the other modules. This is subdivided into

input and output control: jdinput.c controls only input-side processing,

while jdmaster.c handles overall initialization and output-side control.

\* Marker reading: decodes JPEG markers (except for RSTn).

\* Data source manager: supplies the input JPEG datastream. The source

manager supplied with the library knows how to read from a stdio stream

or from a memory buffer; for other behaviors, the surrounding application

may provide its own source manager.

\* Memory manager: same as for compression library.

\* Error handler: same as for compression library.

\* Progress monitor: same as for compression library.

As with compression, the data source manager, error handler, and progress

monitor are candidates for replacement by a surrounding application.

\*\*\* Decompression input and output separation \*\*\*

To support efficient incremental display of progressive JPEG files, the

decompressor is divided into two sections that can run independently:

1. Data input includes marker parsing, entropy decoding, and input into the

coefficient controller's DCT coefficient buffer. Note that this

processing is relatively cheap and fast.

2. Data output reads from the DCT coefficient buffer and performs the IDCT

and all postprocessing steps.

For a progressive JPEG file, the data input processing is allowed to get

arbitrarily far ahead of the data output processing. (This occurs only

if the application calls jpeg\_consume\_input(); otherwise input and output

run in lockstep, since the input section is called only when the output

section needs more data.) In this way the application can avoid making

extra display passes when data is arriving faster than the display pass

can run. Furthermore, it is possible to abort an output pass without

losing anything, since the coefficient buffer is read-only as far as the

output section is concerned. See libjpeg.txt for more detail.

A full-image coefficient array is only created if the JPEG file has multiple

scans (or if the application specifies buffered-image mode anyway). When

reading a single-scan file, the coefficient controller normally creates only

a one-MCU buffer, so input and output processing must run in lockstep in this

case. jpeg\_consume\_input() is effectively a no-op in this situation.

The main impact of dividing the decompressor in this fashion is that we must

be very careful with shared variables in the cinfo data structure. Each

variable that can change during the course of decompression must be

classified as belonging to data input or data output, and each section must

look only at its own variables. For example, the data output section may not

depend on any of the variables that describe the current scan in the JPEG

file, because these may change as the data input section advances into a new

scan.

The progress monitor is (somewhat arbitrarily) defined to treat input of the

file as one pass when buffered-image mode is not used, and to ignore data

input work completely when buffered-image mode is used. Note that the

library has no reliable way to predict the number of passes when dealing

with a progressive JPEG file, nor can it predict the number of output passes

in buffered-image mode. So the work estimate is inherently bogus anyway.

No comparable division is currently made in the compression library, because

there isn't any real need for it.

\*\*\* Data formats \*\*\*

Arrays of pixel sample values use the following data structure:

typedef something JSAMPLE; a pixel component value, 0..MAXJSAMPLE

typedef JSAMPLE \*JSAMPROW; ptr to a row of samples

typedef JSAMPROW \*JSAMPARRAY; ptr to a list of rows

typedef JSAMPARRAY \*JSAMPIMAGE; ptr to a list of color-component arrays

The basic element type JSAMPLE will typically be one of unsigned char,

(signed) char, or short. Short will be used if samples wider than 8 bits are

to be supported (this is a compile-time option). Otherwise, unsigned char is

used if possible. If the compiler only supports signed chars, then it is

necessary to mask off the value when reading. Thus, all reads of JSAMPLE

values must be coded as "GETJSAMPLE(value)", where the macro will be defined

as "((value) & 0xFF)" on signed-char machines and "((int) (value))" elsewhere.

With these conventions, JSAMPLE values can be assumed to be >= 0. This helps

simplify correct rounding during downsampling, etc. The JPEG standard's

specification that sample values run from -128..127 is accommodated by

subtracting 128 from the sample value in the DCT step. Similarly, during

decompression the output of the IDCT step will be immediately shifted back to

0..255. (NB: different values are required when 12-bit samples are in use.

The code is written in terms of MAXJSAMPLE and CENTERJSAMPLE, which will be

defined as 255 and 128 respectively in an 8-bit implementation, and as 4095

and 2048 in a 12-bit implementation.)

We use a pointer per row, rather than a two-dimensional JSAMPLE array. This

choice costs only a small amount of memory and has several benefits:

\* Code using the data structure doesn't need to know the allocated width of

the rows. This simplifies edge expansion/compression, since we can work

in an array that's wider than the logical picture width.

\* Indexing doesn't require multiplication; this is a performance win on many

machines.

\* Arrays with more than 64K total elements can be supported even on machines

where malloc() cannot allocate chunks larger than 64K.

\* The rows forming a component array may be allocated at different times

without extra copying. This trick allows some speedups in smoothing steps

that need access to the previous and next rows.

Note that each color component is stored in a separate array; we don't use the

traditional layout in which the components of a pixel are stored together.

This simplifies coding of modules that work on each component independently,

because they don't need to know how many components there are. Furthermore,

we can read or write each component to a temporary file independently, which

is helpful when dealing with noninterleaved JPEG files.

In general, a specific sample value is accessed by code such as

GETJSAMPLE(image[colorcomponent][row][col])

where col is measured from the image left edge, but row is measured from the

first sample row currently in memory. Either of the first two indexings can

be precomputed by copying the relevant pointer.

Since most image-processing applications prefer to work on images in which

the components of a pixel are stored together, the data passed to or from the

surrounding application uses the traditional convention: a single pixel is

represented by N consecutive JSAMPLE values, and an image row is an array of

(# of color components)\*(image width) JSAMPLEs. One or more rows of data can

be represented by a pointer of type JSAMPARRAY in this scheme. This scheme is

converted to component-wise storage inside the JPEG library. (Applications

that want to skip JPEG preprocessing or postprocessing will have to contend

with component-wise storage.)

Arrays of DCT-coefficient values use the following data structure:

typedef short JCOEF; a 16-bit signed integer

typedef JCOEF JBLOCK[DCTSIZE2]; an 8x8 block of coefficients

typedef JBLOCK \*JBLOCKROW; ptr to one horizontal row of 8x8 blocks

typedef JBLOCKROW \*JBLOCKARRAY; ptr to a list of such rows

typedef JBLOCKARRAY \*JBLOCKIMAGE; ptr to a list of color component arrays

The underlying type is at least a 16-bit signed integer; while "short" is big

enough on all machines of interest, on some machines it is preferable to use

"int" for speed reasons, despite the storage cost. Coefficients are grouped

into 8x8 blocks (but we always use #defines DCTSIZE and DCTSIZE2 rather than

"8" and "64").

The contents of a coefficient block may be in either "natural" or zigzagged

order, and may be true values or divided by the quantization coefficients,

depending on where the block is in the processing pipeline. In the current

library, coefficient blocks are kept in natural order everywhere; the entropy

codecs zigzag or dezigzag the data as it is written or read. The blocks

contain quantized coefficients everywhere outside the DCT/IDCT subsystems.

(This latter decision may need to be revisited to support variable

quantization a la JPEG Part 3.)

Notice that the allocation unit is now a row of 8x8 coefficient blocks,

corresponding to block\_size rows of samples. Otherwise the structure

is much the same as for samples, and for the same reasons.

On machines where malloc() can't handle a request bigger than 64Kb, this data

structure limits us to rows of less than 512 JBLOCKs, or a picture width of

4000+ pixels. This seems an acceptable restriction.

On 80x86 machines, the bottom-level pointer types (JSAMPROW and JBLOCKROW)

must be declared as "far" pointers, but the upper levels can be "near"

(implying that the pointer lists are allocated in the DS segment).

We use a #define symbol FAR, which expands to the "far" keyword when

compiling on 80x86 machines and to nothing elsewhere.

\*\*\* Suspendable processing \*\*\*

In some applications it is desirable to use the JPEG library as an

incremental, memory-to-memory filter. In this situation the data source or

destination may be a limited-size buffer, and we can't rely on being able to

empty or refill the buffer at arbitrary times. Instead the application would

like to have control return from the library at buffer overflow/underrun, and

then resume compression or decompression at a later time.

This scenario is supported for simple cases. (For anything more complex, we

recommend that the application "bite the bullet" and develop real multitasking

capability.) The libjpeg.txt file goes into more detail about the usage and

limitations of this capability; here we address the implications for library

structure.

The essence of the problem is that the entropy codec (coder or decoder) must

be prepared to stop at arbitrary times. In turn, the controllers that call

the entropy codec must be able to stop before having produced or consumed all

the data that they normally would handle in one call. That part is reasonably

straightforward: we make the controller call interfaces include "progress

counters" which indicate the number of data chunks successfully processed, and

we require callers to test the counter rather than just assume all of the data

was processed.

Rather than trying to restart at an arbitrary point, the current Huffman

codecs are designed to restart at the beginning of the current MCU after a

suspension due to buffer overflow/underrun. At the start of each call, the

codec's internal state is loaded from permanent storage (in the JPEG object

structures) into local variables. On successful completion of the MCU, the

permanent state is updated. (This copying is not very expensive, and may even

lead to \*improved\* performance if the local variables can be registerized.)

If a suspension occurs, the codec simply returns without updating the state,

thus effectively reverting to the start of the MCU. Note that this implies

leaving some data unprocessed in the source/destination buffer (ie, the

compressed partial MCU). The data source/destination module interfaces are

specified so as to make this possible. This also implies that the data buffer

must be large enough to hold a worst-case compressed MCU; a couple thousand

bytes should be enough.

In a successive-approximation AC refinement scan, the progressive Huffman

decoder has to be able to undo assignments of newly nonzero coefficients if it

suspends before the MCU is complete, since decoding requires distinguishing

previously-zero and previously-nonzero coefficients. This is a bit tedious

but probably won't have much effect on performance. Other variants of Huffman

decoding need not worry about this, since they will just store the same values

again if forced to repeat the MCU.

This approach would probably not work for an arithmetic codec, since its

modifiable state is quite large and couldn't be copied cheaply. Instead it

would have to suspend and resume exactly at the point of the buffer end.

The JPEG marker reader is designed to cope with suspension at an arbitrary

point. It does so by backing up to the start of the marker parameter segment,

so the data buffer must be big enough to hold the largest marker of interest.

Again, a couple KB should be adequate. (A special "skip" convention is used

to bypass COM and APPn markers, so these can be larger than the buffer size

without causing problems; otherwise a 64K buffer would be needed in the worst

case.)

The JPEG marker writer currently does \*not\* cope with suspension.

We feel that this is not necessary; it is much easier simply to require

the application to ensure there is enough buffer space before starting. (An

empty 2K buffer is more than sufficient for the header markers; and ensuring

there are a dozen or two bytes available before calling jpeg\_finish\_compress()

will suffice for the trailer.) This would not work for writing multi-scan

JPEG files, but we simply do not intend to support that capability with

suspension.

\*\*\* Memory manager services \*\*\*

The JPEG library's memory manager controls allocation and deallocation of

memory, and it manages large "virtual" data arrays on machines where the

operating system does not provide virtual memory. Note that the same

memory manager serves both compression and decompression operations.

In all cases, allocated objects are tied to a particular compression or

decompression master record, and they will be released when that master

record is destroyed.

The memory manager does not provide explicit deallocation of objects.

Instead, objects are created in "pools" of free storage, and a whole pool

can be freed at once. This approach helps prevent storage-leak bugs, and

it speeds up operations whenever malloc/free are slow (as they often are).

The pools can be regarded as lifetime identifiers for objects. Two

pools/lifetimes are defined:

\* JPOOL\_PERMANENT lasts until master record is destroyed

\* JPOOL\_IMAGE lasts until done with image (JPEG datastream)

Permanent lifetime is used for parameters and tables that should be carried

across from one datastream to another; this includes all application-visible

parameters. Image lifetime is used for everything else. (A third lifetime,

JPOOL\_PASS = one processing pass, was originally planned. However it was

dropped as not being worthwhile. The actual usage patterns are such that the

peak memory usage would be about the same anyway; and having per-pass storage

substantially complicates the virtual memory allocation rules --- see below.)

The memory manager deals with three kinds of object:

1. "Small" objects. Typically these require no more than 10K-20K total.

2. "Large" objects. These may require tens to hundreds of K depending on

image size. Semantically they behave the same as small objects, but we

distinguish them for two reasons:

\* On MS-DOS machines, large objects are referenced by FAR pointers,

small objects by NEAR pointers.

\* Pool allocation heuristics may differ for large and small objects.

Note that individual "large" objects cannot exceed the size allowed by

type size\_t, which may be 64K or less on some machines.

3. "Virtual" objects. These are large 2-D arrays of JSAMPLEs or JBLOCKs

(typically large enough for the entire image being processed). The

memory manager provides stripwise access to these arrays. On machines

without virtual memory, the rest of the array may be swapped out to a

temporary file.

(Note: JSAMPARRAY and JBLOCKARRAY data structures are a combination of large

objects for the data proper and small objects for the row pointers. For

convenience and speed, the memory manager provides single routines to create

these structures. Similarly, virtual arrays include a small control block

and a JSAMPARRAY or JBLOCKARRAY working buffer, all created with one call.)

In the present implementation, virtual arrays are only permitted to have image

lifespan. (Permanent lifespan would not be reasonable, and pass lifespan is

not very useful since a virtual array's raison d'etre is to store data for

multiple passes through the image.) We also expect that only "small" objects

will be given permanent lifespan, though this restriction is not required by

the memory manager.

In a non-virtual-memory machine, some performance benefit can be gained by

making the in-memory buffers for virtual arrays be as large as possible.

(For small images, the buffers might fit entirely in memory, so blind

swapping would be very wasteful.) The memory manager will adjust the height

of the buffers to fit within a prespecified maximum memory usage. In order

to do this in a reasonably optimal fashion, the manager needs to allocate all

of the virtual arrays at once. Therefore, there isn't a one-step allocation

routine for virtual arrays; instead, there is a "request" routine that simply

allocates the control block, and a "realize" routine (called just once) that

determines space allocation and creates all of the actual buffers. The

realize routine must allow for space occupied by non-virtual large objects.

(We don't bother to factor in the space needed for small objects, on the

grounds that it isn't worth the trouble.)

To support all this, we establish the following protocol for doing business

with the memory manager:

1. Modules must request virtual arrays (which may have only image lifespan)

during the initial setup phase, i.e., in their jinit\_xxx routines.

2. All "large" objects (including JSAMPARRAYs and JBLOCKARRAYs) must also be

allocated during initial setup.

3. realize\_virt\_arrays will be called at the completion of initial setup.

The above conventions ensure that sufficient information is available

for it to choose a good size for virtual array buffers.

Small objects of any lifespan may be allocated at any time. We expect that

the total space used for small objects will be small enough to be negligible

in the realize\_virt\_arrays computation.

In a virtual-memory machine, we simply pretend that the available space is

infinite, thus causing realize\_virt\_arrays to decide that it can allocate all

the virtual arrays as full-size in-memory buffers. The overhead of the

virtual-array access protocol is very small when no swapping occurs.

A virtual array can be specified to be "pre-zeroed"; when this flag is set,

never-yet-written sections of the array are set to zero before being made

available to the caller. If this flag is not set, never-written sections

of the array contain garbage. (This feature exists primarily because the

equivalent logic would otherwise be needed in jdcoefct.c for progressive

JPEG mode; we may as well make it available for possible other uses.)

The first write pass on a virtual array is required to occur in top-to-bottom

order; read passes, as well as any write passes after the first one, may

access the array in any order. This restriction exists partly to simplify

the virtual array control logic, and partly because some file systems may not

support seeking beyond the current end-of-file in a temporary file. The main

implication of this restriction is that rearrangement of rows (such as

converting top-to-bottom data order to bottom-to-top) must be handled while

reading data out of the virtual array, not while putting it in.

\*\*\* Memory manager internal structure \*\*\*

To isolate system dependencies as much as possible, we have broken the

memory manager into two parts. There is a reasonably system-independent

"front end" (jmemmgr.c) and a "back end" that contains only the code

likely to change across systems. All of the memory management methods

outlined above are implemented by the front end. The back end provides

the following routines for use by the front end (none of these routines

are known to the rest of the JPEG code):

jpeg\_mem\_init, jpeg\_mem\_term system-dependent initialization/shutdown

jpeg\_get\_small, jpeg\_free\_small interface to malloc and free library routines

(or their equivalents)

jpeg\_get\_large, jpeg\_free\_large interface to FAR malloc/free in MSDOS machines;

else usually the same as

jpeg\_get\_small/jpeg\_free\_small

jpeg\_mem\_available estimate available memory

jpeg\_open\_backing\_store create a backing-store object

read\_backing\_store, manipulate a backing-store object

write\_backing\_store,

close\_backing\_store

On some systems there will be more than one type of backing-store object

(specifically, in MS-DOS a backing store file might be an area of extended

memory as well as a disk file). jpeg\_open\_backing\_store is responsible for

choosing how to implement a given object. The read/write/close routines

are method pointers in the structure that describes a given object; this

lets them be different for different object types.

It may be necessary to ensure that backing store objects are explicitly

released upon abnormal program termination. For example, MS-DOS won't free

extended memory by itself. To support this, we will expect the main program

or surrounding application to arrange to call self\_destruct (typically via

jpeg\_destroy) upon abnormal termination. This may require a SIGINT signal

handler or equivalent. We don't want to have the back end module install its

own signal handler, because that would pre-empt the surrounding application's

ability to control signal handling.

The IJG distribution includes several memory manager back end implementations.

Usually the same back end should be suitable for all applications on a given

system, but it is possible for an application to supply its own back end at

need.

\*\*\* Implications of DNL marker \*\*\*

Some JPEG files may use a DNL marker to postpone definition of the image

height (this would be useful for a fax-like scanner's output, for instance).

In these files the SOF marker claims the image height is 0, and you only

find out the true image height at the end of the first scan.

We could read these files as follows:

1. Upon seeing zero image height, replace it by 65535 (the maximum allowed).

2. When the DNL is found, update the image height in the global image

descriptor.

This implies that control modules must avoid making copies of the image

height, and must re-test for termination after each MCU row. This would

be easy enough to do.

In cases where image-size data structures are allocated, this approach will

result in very inefficient use of virtual memory or much-larger-than-necessary

temporary files. This seems acceptable for something that probably won't be a

mainstream usage. People might have to forgo use of memory-hogging options

(such as two-pass color quantization or noninterleaved JPEG files) if they

want efficient conversion of such files. (One could improve efficiency by

demanding a user-supplied upper bound for the height, less than 65536; in most

cases it could be much less.)

The standard also permits the SOF marker to overestimate the image height,

with a DNL to give the true, smaller height at the end of the first scan.

This would solve the space problems if the overestimate wasn't too great.

However, it implies that you don't even know whether DNL will be used.

This leads to a couple of very serious objections:

1. Testing for a DNL marker must occur in the inner loop of the decompressor's

Huffman decoder; this implies a speed penalty whether the feature is used

or not.

2. There is no way to hide the last-minute change in image height from an

application using the decoder. Thus \*every\* application using the IJG

library would suffer a complexity penalty whether it cared about DNL or

not.

We currently do not support DNL because of these problems.

A different approach is to insist that DNL-using files be preprocessed by a

separate program that reads ahead to the DNL, then goes back and fixes the SOF

marker. This is a much simpler solution and is probably far more efficient.

Even if one wants piped input, buffering the first scan of the JPEG file needs

a lot smaller temp file than is implied by the maximum-height method. For

this approach we'd simply treat DNL as a no-op in the decompressor (at most,

check that it matches the SOF image height).

We will not worry about making the compressor capable of outputting DNL.

Something similar to the first scheme above could be applied if anyone ever

wants to make that work.