USING THE IJG JPEG LIBRARY

Copyright (C) 1994-1998, Thomas G. Lane.

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

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

This file describes how to use the IJG JPEG library within an application

program. Read it if you want to write a program that uses the library.

The file example.c provides heavily commented skeleton code for calling the

JPEG library. Also see jpeglib.h (the include file to be used by application

programs) for full details about data structures and function parameter lists.

The library source code, of course, is the ultimate reference.

Note that there have been \*major\* changes from the application interface

presented by IJG version 4 and earlier versions. The old design had several

inherent limitations, and it had accumulated a lot of cruft as we added

features while trying to minimize application-interface changes. We have

sacrificed backward compatibility in the version 5 rewrite, but we think the

improvements justify this.

TABLE OF CONTENTS

-----------------

Overview:

Functions provided by the library

Outline of typical usage

Basic library usage:

Data formats

Compression details

Decompression details

Mechanics of usage: include files, linking, etc

Advanced features:

Compression parameter selection

Decompression parameter selection

Special color spaces

Error handling

Compressed data handling (source and destination managers)

I/O suspension

Progressive JPEG support

Buffered-image mode

Abbreviated datastreams and multiple images

Special markers

Raw (downsampled) image data

Really raw data: DCT coefficients

Progress monitoring

Memory management

Memory usage

Library compile-time options

Portability considerations

Notes for MS-DOS implementors

You should read at least the overview and basic usage sections before trying

to program with the library. The sections on advanced features can be read

if and when you need them.

OVERVIEW

========

Functions provided by the library

---------------------------------

The IJG JPEG library provides C code to read and write JPEG-compressed image

files. The surrounding application program receives or supplies image data a

scanline at a time, using a straightforward uncompressed image format. All

details of color conversion and other preprocessing/postprocessing can be

handled by the library.

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. The application indirectly selects use of this code

by specifying the format in which it wishes to supply or receive image data.

For example, if colormapped output is requested, then the decompression

library automatically invokes color quantization.

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.

A word about functions \*not\* provided by the library. We handle a subset of

the ISO JPEG standard; most baseline, extended-sequential, and progressive

JPEG processes are supported. (Our subset includes all features now in common

use.) Unsupported ISO options include:

\* Hierarchical storage

\* Lossless JPEG

\* Arithmetic entropy coding (unsupported for legal reasons)

\* DNL marker

\* Nonintegral subsampling ratios

We support both 8- and 12-bit data precision, but this is a compile-time

choice rather than a run-time choice; hence it is difficult to use both

precisions in a single application.

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, this library is

used by the free LIBTIFF library to support JPEG compression in TIFF.)

Outline of typical usage

------------------------

The rough outline of a JPEG compression operation is:

Allocate and initialize a JPEG compression object

Specify the destination for the compressed data (eg, a file)

Set parameters for compression, including image size & colorspace

jpeg\_start\_compress(...);

while (scan lines remain to be written)

jpeg\_write\_scanlines(...);

jpeg\_finish\_compress(...);

Release the JPEG compression object

A JPEG compression object holds parameters and working state for the JPEG

library. We make creation/destruction of the object separate from starting

or finishing compression of an image; the same object can be re-used for a

series of image compression operations. This makes it easy to re-use the

same parameter settings for a sequence of images. Re-use of a JPEG object

also has important implications for processing abbreviated JPEG datastreams,

as discussed later.

The image data to be compressed is supplied to jpeg\_write\_scanlines() from

in-memory buffers. If the application is doing file-to-file compression,

reading image data from the source file is the application's responsibility.

The library emits compressed data by calling a "data destination manager",

which typically will write the data into a file; but the application can

provide its own destination manager to do something else.

Similarly, the rough outline of a JPEG decompression operation is:

Allocate and initialize a JPEG decompression object

Specify the source of the compressed data (eg, a file)

Call jpeg\_read\_header() to obtain image info

Set parameters for decompression

jpeg\_start\_decompress(...);

while (scan lines remain to be read)

jpeg\_read\_scanlines(...);

jpeg\_finish\_decompress(...);

Release the JPEG decompression object

This is comparable to the compression outline except that reading the

datastream header is a separate step. This is helpful because information

about the image's size, colorspace, etc is available when the application

selects decompression parameters. For example, the application can choose an

output scaling ratio that will fit the image into the available screen size.

The decompression library obtains compressed data by calling a data source

manager, which typically will read the data from a file; but other behaviors

can be obtained with a custom source manager. Decompressed data is delivered

into in-memory buffers passed to jpeg\_read\_scanlines().

It is possible to abort an incomplete compression or decompression operation

by calling jpeg\_abort(); or, if you do not need to retain the JPEG object,

simply release it by calling jpeg\_destroy().

JPEG compression and decompression objects are two separate struct types.

However, they share some common fields, and certain routines such as

jpeg\_destroy() can work on either type of object.

The JPEG library has no static variables: all state is in the compression

or decompression object. Therefore it is possible to process multiple

compression and decompression operations concurrently, using multiple JPEG

objects.

Both compression and decompression can be done in an incremental memory-to-

memory fashion, if suitable source/destination managers are used. See the

section on "I/O suspension" for more details.

BASIC LIBRARY USAGE

===================

Data formats

------------

Before diving into procedural details, it is helpful to understand the

image data format that the JPEG library expects or returns.

The standard input image format is a rectangular array of pixels, with each

pixel having the same number of "component" or "sample" values (color

channels). You must specify how many components there are and the colorspace

interpretation of the components. Most applications will use RGB data

(three components per pixel) or grayscale data (one component per pixel).

PLEASE NOTE THAT RGB DATA IS THREE SAMPLES PER PIXEL, GRAYSCALE ONLY ONE.

A remarkable number of people manage to miss this, only to find that their

programs don't work with grayscale JPEG files.

There is no provision for colormapped input. JPEG files are always full-color

or full grayscale (or sometimes another colorspace such as CMYK). You can

feed in a colormapped image by expanding it to full-color format. However

JPEG often doesn't work very well with source data that has been colormapped,

because of dithering noise. This is discussed in more detail in the JPEG FAQ

and the other references mentioned in the README file.

Pixels are stored by scanlines, with each scanline running from left to

right. The component values for each pixel are adjacent in the row; for

example, R,G,B,R,G,B,R,G,B,... for 24-bit RGB color. Each scanline is an

array of data type JSAMPLE --- which is typically "unsigned char", unless

you've changed jmorecfg.h. (You can also change the RGB pixel layout, say

to B,G,R order, by modifying jmorecfg.h. But see the restrictions listed in

that file before doing so.)

A 2-D array of pixels is formed by making a list of pointers to the starts of

scanlines; so the scanlines need not be physically adjacent in memory. Even

if you process just one scanline at a time, you must make a one-element

pointer array to conform to this structure. Pointers to JSAMPLE rows are of

type JSAMPROW, and the pointer to the pointer array is of type JSAMPARRAY.

The library accepts or supplies one or more complete scanlines per call.

It is not possible to process part of a row at a time. Scanlines are always

processed top-to-bottom. You can process an entire image in one call if you

have it all in memory, but usually it's simplest to process one scanline at

a time.

For best results, source data values should have the precision specified by

BITS\_IN\_JSAMPLE (normally 8 bits). For instance, if you choose to compress

data that's only 6 bits/channel, you should left-justify each value in a

byte before passing it to the compressor. If you need to compress data

that has more than 8 bits/channel, compile with BITS\_IN\_JSAMPLE = 12.

(See "Library compile-time options", later.)

The data format returned by the decompressor is the same in all details,

except that colormapped output is supported. (Again, a JPEG file is never

colormapped. But you can ask the decompressor to perform on-the-fly color

quantization to deliver colormapped output.) If you request colormapped

output then the returned data array contains a single JSAMPLE per pixel;

its value is an index into a color map. The color map is represented as

a 2-D JSAMPARRAY in which each row holds the values of one color component,

that is, colormap[i][j] is the value of the i'th color component for pixel

value (map index) j. Note that since the colormap indexes are stored in

JSAMPLEs, the maximum number of colors is limited by the size of JSAMPLE

(ie, at most 256 colors for an 8-bit JPEG library).

Compression details

-------------------

Here we revisit the JPEG compression outline given in the overview.

1. Allocate and initialize a JPEG compression object.

A JPEG compression object is a "struct jpeg\_compress\_struct". (It also has

a bunch of subsidiary structures which are allocated via malloc(), but the

application doesn't control those directly.) This struct can be just a local

variable in the calling routine, if a single routine is going to execute the

whole JPEG compression sequence. Otherwise it can be static or allocated

from malloc().

You will also need a structure representing a JPEG error handler. The part

of this that the library cares about is a "struct jpeg\_error\_mgr". If you

are providing your own error handler, you'll typically want to embed the

jpeg\_error\_mgr struct in a larger structure; this is discussed later under

"Error handling". For now we'll assume you are just using the default error

handler. The default error handler will print JPEG error/warning messages

on stderr, and it will call exit() if a fatal error occurs.

You must initialize the error handler structure, store a pointer to it into

the JPEG object's "err" field, and then call jpeg\_create\_compress() to

initialize the rest of the JPEG object.

Typical code for this step, if you are using the default error handler, is

struct jpeg\_compress\_struct cinfo;

struct jpeg\_error\_mgr jerr;

...

cinfo.err = jpeg\_std\_error(&jerr);

jpeg\_create\_compress(&cinfo);

jpeg\_create\_compress allocates a small amount of memory, so it could fail

if you are out of memory. In that case it will exit via the error handler;

that's why the error handler must be initialized first.

2. Specify the destination for the compressed data (eg, a file).

As previously mentioned, the JPEG library delivers compressed data to a

"data destination" module. The library includes one data destination

module which knows how to write to a stdio stream. You can use your own

destination module if you want to do something else, as discussed later.

If you use the standard destination module, you must open the target stdio

stream beforehand. Typical code for this step looks like:

FILE \* outfile;

...

if ((outfile = fopen(filename, "wb")) == NULL) {

fprintf(stderr, "can't open %s\n", filename);

exit(1);

}

jpeg\_stdio\_dest(&cinfo, outfile);

where the last line invokes the standard destination module.

WARNING: it is critical that the binary compressed data be delivered to the

output file unchanged. On non-Unix systems the stdio library may perform

newline translation or otherwise corrupt binary data. To suppress this

behavior, you may need to use a "b" option to fopen (as shown above), or use

setmode() or another routine to put the stdio stream in binary mode. See

cjpeg.c and djpeg.c for code that has been found to work on many systems.

You can select the data destination after setting other parameters (step 3),

if that's more convenient. You may not change the destination between

calling jpeg\_start\_compress() and jpeg\_finish\_compress().

3. Set parameters for compression, including image size & colorspace.

You must supply information about the source image by setting the following

fields in the JPEG object (cinfo structure):

image\_width Width of image, in pixels

image\_height Height of image, in pixels

input\_components Number of color channels (samples per pixel)

in\_color\_space Color space of source image

The image dimensions are, hopefully, obvious. JPEG supports image dimensions

of 1 to 64K pixels in either direction. The input color space is typically

RGB or grayscale, and input\_components is 3 or 1 accordingly. (See "Special

color spaces", later, for more info.) The in\_color\_space field must be

assigned one of the J\_COLOR\_SPACE enum constants, typically JCS\_RGB or

JCS\_GRAYSCALE.

JPEG has a large number of compression parameters that determine how the

image is encoded. Most applications don't need or want to know about all

these parameters. You can set all the parameters to reasonable defaults by

calling jpeg\_set\_defaults(); then, if there are particular values you want

to change, you can do so after that. The "Compression parameter selection"

section tells about all the parameters.

You must set in\_color\_space correctly before calling jpeg\_set\_defaults(),

because the defaults depend on the source image colorspace. However the

other three source image parameters need not be valid until you call

jpeg\_start\_compress(). There's no harm in calling jpeg\_set\_defaults() more

than once, if that happens to be convenient.

Typical code for a 24-bit RGB source image is

cinfo.image\_width = Width; /\* image width and height, in pixels \*/

cinfo.image\_height = Height;

cinfo.input\_components = 3; /\* # of color components per pixel \*/

cinfo.in\_color\_space = JCS\_RGB; /\* colorspace of input image \*/

jpeg\_set\_defaults(&cinfo);

/\* Make optional parameter settings here \*/

4. jpeg\_start\_compress(...);

After you have established the data destination and set all the necessary

source image info and other parameters, call jpeg\_start\_compress() to begin

a compression cycle. This will initialize internal state, allocate working

storage, and emit the first few bytes of the JPEG datastream header.

Typical code:

jpeg\_start\_compress(&cinfo, TRUE);

The "TRUE" parameter ensures that a complete JPEG interchange datastream

will be written. This is appropriate in most cases. If you think you might

want to use an abbreviated datastream, read the section on abbreviated

datastreams, below.

Once you have called jpeg\_start\_compress(), you may not alter any JPEG

parameters or other fields of the JPEG object until you have completed

the compression cycle.

5. while (scan lines remain to be written)

jpeg\_write\_scanlines(...);

Now write all the required image data by calling jpeg\_write\_scanlines()

one or more times. You can pass one or more scanlines in each call, up

to the total image height. In most applications it is convenient to pass

just one or a few scanlines at a time. The expected format for the passed

data is discussed under "Data formats", above.

Image data should be written in top-to-bottom scanline order. The JPEG spec

contains some weasel wording about how top and bottom are application-defined

terms (a curious interpretation of the English language...) but if you want

your files to be compatible with everyone else's, you WILL use top-to-bottom

order. If the source data must be read in bottom-to-top order, you can use

the JPEG library's virtual array mechanism to invert the data efficiently.

Examples of this can be found in the sample application cjpeg.

The library maintains a count of the number of scanlines written so far

in the next\_scanline field of the JPEG object. Usually you can just use

this variable as the loop counter, so that the loop test looks like

"while (cinfo.next\_scanline < cinfo.image\_height)".

Code for this step depends heavily on the way that you store the source data.

example.c shows the following code for the case of a full-size 2-D source

array containing 3-byte RGB pixels:

JSAMPROW row\_pointer[1]; /\* pointer to a single row \*/

int row\_stride; /\* physical row width in buffer \*/

row\_stride = image\_width \* 3; /\* JSAMPLEs per row in image\_buffer \*/

while (cinfo.next\_scanline < cinfo.image\_height) {

row\_pointer[0] = & image\_buffer[cinfo.next\_scanline \* row\_stride];

jpeg\_write\_scanlines(&cinfo, row\_pointer, 1);

}

jpeg\_write\_scanlines() returns the number of scanlines actually written.

This will normally be equal to the number passed in, so you can usually

ignore the return value. It is different in just two cases:

\* If you try to write more scanlines than the declared image height,

the additional scanlines are ignored.

\* If you use a suspending data destination manager, output buffer overrun

will cause the compressor to return before accepting all the passed lines.

This feature is discussed under "I/O suspension", below. The normal

stdio destination manager will NOT cause this to happen.

In any case, the return value is the same as the change in the value of

next\_scanline.

6. jpeg\_finish\_compress(...);

After all the image data has been written, call jpeg\_finish\_compress() to

complete the compression cycle. This step is ESSENTIAL to ensure that the

last bufferload of data is written to the data destination.

jpeg\_finish\_compress() also releases working memory associated with the JPEG

object.

Typical code:

jpeg\_finish\_compress(&cinfo);

If using the stdio destination manager, don't forget to close the output

stdio stream (if necessary) afterwards.

If you have requested a multi-pass operating mode, such as Huffman code

optimization, jpeg\_finish\_compress() will perform the additional passes using

data buffered by the first pass. In this case jpeg\_finish\_compress() may take

quite a while to complete. With the default compression parameters, this will

not happen.

It is an error to call jpeg\_finish\_compress() before writing the necessary

total number of scanlines. If you wish to abort compression, call

jpeg\_abort() as discussed below.

After completing a compression cycle, you may dispose of the JPEG object

as discussed next, or you may use it to compress another image. In that case

return to step 2, 3, or 4 as appropriate. If you do not change the

destination manager, the new datastream will be written to the same target.

If you do not change any JPEG parameters, the new datastream will be written

with the same parameters as before. Note that you can change the input image

dimensions freely between cycles, but if you change the input colorspace, you

should call jpeg\_set\_defaults() to adjust for the new colorspace; and then

you'll need to repeat all of step 3.

7. Release the JPEG compression object.

When you are done with a JPEG compression object, destroy it by calling

jpeg\_destroy\_compress(). This will free all subsidiary memory (regardless of

the previous state of the object). Or you can call jpeg\_destroy(), which

works for either compression or decompression objects --- this may be more

convenient if you are sharing code between compression and decompression

cases. (Actually, these routines are equivalent except for the declared type

of the passed pointer. To avoid gripes from ANSI C compilers, jpeg\_destroy()

should be passed a j\_common\_ptr.)

If you allocated the jpeg\_compress\_struct structure from malloc(), freeing

it is your responsibility --- jpeg\_destroy() won't. Ditto for the error

handler structure.

Typical code:

jpeg\_destroy\_compress(&cinfo);

8. Aborting.

If you decide to abort a compression cycle before finishing, you can clean up

in either of two ways:

\* If you don't need the JPEG object any more, just call

jpeg\_destroy\_compress() or jpeg\_destroy() to release memory. This is

legitimate at any point after calling jpeg\_create\_compress() --- in fact,

it's safe even if jpeg\_create\_compress() fails.

\* If you want to re-use the JPEG object, call jpeg\_abort\_compress(), or call

jpeg\_abort() which works on both compression and decompression objects.

This will return the object to an idle state, releasing any working memory.

jpeg\_abort() is allowed at any time after successful object creation.

Note that cleaning up the data destination, if required, is your

responsibility; neither of these routines will call term\_destination().

(See "Compressed data handling", below, for more about that.)

jpeg\_destroy() and jpeg\_abort() are the only safe calls to make on a JPEG

object that has reported an error by calling error\_exit (see "Error handling"

for more info). The internal state of such an object is likely to be out of

whack. Either of these two routines will return the object to a known state.

Decompression details

---------------------

Here we revisit the JPEG decompression outline given in the overview.

1. Allocate and initialize a JPEG decompression object.

This is just like initialization for compression, as discussed above,

except that the object is a "struct jpeg\_decompress\_struct" and you

call jpeg\_create\_decompress(). Error handling is exactly the same.

Typical code:

struct jpeg\_decompress\_struct cinfo;

struct jpeg\_error\_mgr jerr;

...

cinfo.err = jpeg\_std\_error(&jerr);

jpeg\_create\_decompress(&cinfo);

(Both here and in the IJG code, we usually use variable name "cinfo" for

both compression and decompression objects.)

2. Specify the source of the compressed data (eg, a file).

As previously mentioned, the JPEG library reads compressed data from a "data

source" module. The library includes one data source module which knows how

to read from a stdio stream. You can use your own source module if you want

to do something else, as discussed later.

If you use the standard source module, you must open the source stdio stream

beforehand. Typical code for this step looks like:

FILE \* infile;

...

if ((infile = fopen(filename, "rb")) == NULL) {

fprintf(stderr, "can't open %s\n", filename);

exit(1);

}

jpeg\_stdio\_src(&cinfo, infile);

where the last line invokes the standard source module.

WARNING: it is critical that the binary compressed data be read unchanged.

On non-Unix systems the stdio library may perform newline translation or

otherwise corrupt binary data. To suppress this behavior, you may need to use

a "b" option to fopen (as shown above), or use setmode() or another routine to

put the stdio stream in binary mode. See cjpeg.c and djpeg.c for code that

has been found to work on many systems.

You may not change the data source between calling jpeg\_read\_header() and

jpeg\_finish\_decompress(). If you wish to read a series of JPEG images from

a single source file, you should repeat the jpeg\_read\_header() to

jpeg\_finish\_decompress() sequence without reinitializing either the JPEG

object or the data source module; this prevents buffered input data from

being discarded.

3. Call jpeg\_read\_header() to obtain image info.

Typical code for this step is just

jpeg\_read\_header(&cinfo, TRUE);

This will read the source datastream header markers, up to the beginning

of the compressed data proper. On return, the image dimensions and other

info have been stored in the JPEG object. The application may wish to

consult this information before selecting decompression parameters.

More complex code is necessary if

\* A suspending data source is used --- in that case jpeg\_read\_header()

may return before it has read all the header data. See "I/O suspension",

below. The normal stdio source manager will NOT cause this to happen.

\* Abbreviated JPEG files are to be processed --- see the section on

abbreviated datastreams. Standard applications that deal only in

interchange JPEG files need not be concerned with this case either.

It is permissible to stop at this point if you just wanted to find out the

image dimensions and other header info for a JPEG file. In that case,

call jpeg\_destroy() when you are done with the JPEG object, or call

jpeg\_abort() to return it to an idle state before selecting a new data

source and reading another header.

4. Set parameters for decompression.

jpeg\_read\_header() sets appropriate default decompression parameters based on

the properties of the image (in particular, its colorspace). However, you

may well want to alter these defaults before beginning the decompression.

For example, the default is to produce full color output from a color file.

If you want colormapped output you must ask for it. Other options allow the

returned image to be scaled and allow various speed/quality tradeoffs to be

selected. "Decompression parameter selection", below, gives details.

If the defaults are appropriate, nothing need be done at this step.

Note that all default values are set by each call to jpeg\_read\_header().

If you reuse a decompression object, you cannot expect your parameter

settings to be preserved across cycles, as you can for compression.

You must set desired parameter values each time.

5. jpeg\_start\_decompress(...);

Once the parameter values are satisfactory, call jpeg\_start\_decompress() to

begin decompression. This will initialize internal state, allocate working

memory, and prepare for returning data.

Typical code is just

jpeg\_start\_decompress(&cinfo);

If you have requested a multi-pass operating mode, such as 2-pass color

quantization, jpeg\_start\_decompress() will do everything needed before data

output can begin. In this case jpeg\_start\_decompress() may take quite a while

to complete. With a single-scan (non progressive) JPEG file and default

decompression parameters, this will not happen; jpeg\_start\_decompress() will

return quickly.

After this call, the final output image dimensions, including any requested

scaling, are available in the JPEG object; so is the selected colormap, if

colormapped output has been requested. Useful fields include

output\_width image width and height, as scaled

output\_height

out\_color\_components # of color components in out\_color\_space

output\_components # of color components returned per pixel

colormap the selected colormap, if any

actual\_number\_of\_colors number of entries in colormap

output\_components is 1 (a colormap index) when quantizing colors; otherwise it

equals out\_color\_components. It is the number of JSAMPLE values that will be

emitted per pixel in the output arrays.

Typically you will need to allocate data buffers to hold the incoming image.

You will need output\_width \* output\_components JSAMPLEs per scanline in your

output buffer, and a total of output\_height scanlines will be returned.

Note: if you are using the JPEG library's internal memory manager to allocate

data buffers (as djpeg does), then the manager's protocol requires that you

request large buffers \*before\* calling jpeg\_start\_decompress(). This is a

little tricky since the output\_XXX fields are not normally valid then. You

can make them valid by calling jpeg\_calc\_output\_dimensions() after setting the

relevant parameters (scaling, output color space, and quantization flag).

6. while (scan lines remain to be read)

jpeg\_read\_scanlines(...);

Now you can read the decompressed image data by calling jpeg\_read\_scanlines()

one or more times. At each call, you pass in the maximum number of scanlines

to be read (ie, the height of your working buffer); jpeg\_read\_scanlines()

will return up to that many lines. The return value is the number of lines

actually read. The format of the returned data is discussed under "Data

formats", above. Don't forget that grayscale and color JPEGs will return

different data formats!

Image data is returned in top-to-bottom scanline order. If you must write

out the image in bottom-to-top order, you can use the JPEG library's virtual

array mechanism to invert the data efficiently. Examples of this can be

found in the sample application djpeg.

The library maintains a count of the number of scanlines returned so far

in the output\_scanline field of the JPEG object. Usually you can just use

this variable as the loop counter, so that the loop test looks like

"while (cinfo.output\_scanline < cinfo.output\_height)". (Note that the test

should NOT be against image\_height, unless you never use scaling. The

image\_height field is the height of the original unscaled image.)

The return value always equals the change in the value of output\_scanline.

If you don't use a suspending data source, it is safe to assume that

jpeg\_read\_scanlines() reads at least one scanline per call, until the

bottom of the image has been reached.

If you use a buffer larger than one scanline, it is NOT safe to assume that

jpeg\_read\_scanlines() fills it. (The current implementation returns only a

few scanlines per call, no matter how large a buffer you pass.) So you must

always provide a loop that calls jpeg\_read\_scanlines() repeatedly until the

whole image has been read.

7. jpeg\_finish\_decompress(...);

After all the image data has been read, call jpeg\_finish\_decompress() to

complete the decompression cycle. This causes working memory associated

with the JPEG object to be released.

Typical code:

jpeg\_finish\_decompress(&cinfo);

If using the stdio source manager, don't forget to close the source stdio

stream if necessary.

It is an error to call jpeg\_finish\_decompress() before reading the correct

total number of scanlines. If you wish to abort decompression, call

jpeg\_abort() as discussed below.

After completing a decompression cycle, you may dispose of the JPEG object as

discussed next, or you may use it to decompress another image. In that case

return to step 2 or 3 as appropriate. If you do not change the source

manager, the next image will be read from the same source.

8. Release the JPEG decompression object.

When you are done with a JPEG decompression object, destroy it by calling

jpeg\_destroy\_decompress() or jpeg\_destroy(). The previous discussion of

destroying compression objects applies here too.

Typical code:

jpeg\_destroy\_decompress(&cinfo);

9. Aborting.

You can abort a decompression cycle by calling jpeg\_destroy\_decompress() or

jpeg\_destroy() if you don't need the JPEG object any more, or

jpeg\_abort\_decompress() or jpeg\_abort() if you want to reuse the object.

The previous discussion of aborting compression cycles applies here too.

Mechanics of usage: include files, linking, etc

-----------------------------------------------

Applications using the JPEG library should include the header file jpeglib.h

to obtain declarations of data types and routines. Before including

jpeglib.h, include system headers that define at least the typedefs FILE and

size\_t. On ANSI-conforming systems, including <stdio.h> is sufficient; on

older Unix systems, you may need <sys/types.h> to define size\_t.

If the application needs to refer to individual JPEG library error codes, also

include jerror.h to define those symbols.

jpeglib.h indirectly includes the files jconfig.h and jmorecfg.h. If you are

installing the JPEG header files in a system directory, you will want to

install all four files: jpeglib.h, jerror.h, jconfig.h, jmorecfg.h.

The most convenient way to include the JPEG code into your executable program

is to prepare a library file ("libjpeg.a", or a corresponding name on non-Unix

machines) and reference it at your link step. If you use only half of the

library (only compression or only decompression), only that much code will be

included from the library, unless your linker is hopelessly brain-damaged.

The supplied makefiles build libjpeg.a automatically (see install.doc).

While you can build the JPEG library as a shared library if the whim strikes

you, we don't really recommend it. The trouble with shared libraries is that

at some point you'll probably try to substitute a new version of the library

without recompiling the calling applications. That generally doesn't work

because the parameter struct declarations usually change with each new

version. In other words, the library's API is \*not\* guaranteed binary

compatible across versions; we only try to ensure source-code compatibility.

(In hindsight, it might have been smarter to hide the parameter structs from

applications and introduce a ton of access functions instead. Too late now,

however.)

On some systems your application may need to set up a signal handler to ensure

that temporary files are deleted if the program is interrupted. This is most

critical if you are on MS-DOS and use the jmemdos.c memory manager back end;

it will try to grab extended memory for temp files, and that space will NOT be

freed automatically. See cjpeg.c or djpeg.c for an example signal handler.

It may be worth pointing out that the core JPEG library does not actually

require the stdio library: only the default source/destination managers and

error handler need it. You can use the library in a stdio-less environment

if you replace those modules and use jmemnobs.c (or another memory manager of

your own devising). More info about the minimum system library requirements

may be found in jinclude.h.

ADVANCED FEATURES

=================

Compression parameter selection

-------------------------------

This section describes all the optional parameters you can set for JPEG

compression, as well as the "helper" routines provided to assist in this

task. Proper setting of some parameters requires detailed understanding

of the JPEG standard; if you don't know what a parameter is for, it's best

not to mess with it! See REFERENCES in the README file for pointers to

more info about JPEG.

It's a good idea to call jpeg\_set\_defaults() first, even if you plan to set

all the parameters; that way your code is more likely to work with future JPEG

libraries that have additional parameters. For the same reason, we recommend

you use a helper routine where one is provided, in preference to twiddling

cinfo fields directly.

The helper routines are:

jpeg\_set\_defaults (j\_compress\_ptr cinfo)

This routine sets all JPEG parameters to reasonable defaults, using

only the input image's color space (field in\_color\_space, which must

already be set in cinfo). Many applications will only need to use

this routine and perhaps jpeg\_set\_quality().

jpeg\_set\_colorspace (j\_compress\_ptr cinfo, J\_COLOR\_SPACE colorspace)

Sets the JPEG file's colorspace (field jpeg\_color\_space) as specified,

and sets other color-space-dependent parameters appropriately. See

"Special color spaces", below, before using this. A large number of

parameters, including all per-component parameters, are set by this

routine; if you want to twiddle individual parameters you should call

jpeg\_set\_colorspace() before rather than after.

jpeg\_default\_colorspace (j\_compress\_ptr cinfo)

Selects an appropriate JPEG colorspace based on cinfo->in\_color\_space,

and calls jpeg\_set\_colorspace(). This is actually a subroutine of

jpeg\_set\_defaults(). It's broken out in case you want to change

just the colorspace-dependent JPEG parameters.

jpeg\_set\_quality (j\_compress\_ptr cinfo, int quality, boolean force\_baseline)

Constructs JPEG quantization tables appropriate for the indicated

quality setting. The quality value is expressed on the 0..100 scale

recommended by IJG (cjpeg's "-quality" switch uses this routine).

Note that the exact mapping from quality values to tables may change

in future IJG releases as more is learned about DCT quantization.

If the force\_baseline parameter is TRUE, then the quantization table

entries are constrained to the range 1..255 for full JPEG baseline

compatibility. In the current implementation, this only makes a

difference for quality settings below 25, and it effectively prevents

very small/low quality files from being generated. The IJG decoder

is capable of reading the non-baseline files generated at low quality

settings when force\_baseline is FALSE, but other decoders may not be.

jpeg\_set\_linear\_quality (j\_compress\_ptr cinfo, int scale\_factor,

boolean force\_baseline)

Same as jpeg\_set\_quality() except that the generated tables are the

sample tables given in the JPEC spec section K.1, multiplied by the

specified scale factor (which is expressed as a percentage; thus

scale\_factor = 100 reproduces the spec's tables). Note that larger

scale factors give lower quality. This entry point is useful for

conforming to the Adobe PostScript DCT conventions, but we do not

recommend linear scaling as a user-visible quality scale otherwise.

force\_baseline again constrains the computed table entries to 1..255.

int jpeg\_quality\_scaling (int quality)

Converts a value on the IJG-recommended quality scale to a linear

scaling percentage. Note that this routine may change or go away

in future releases --- IJG may choose to adopt a scaling method that

can't be expressed as a simple scalar multiplier, in which case the

premise of this routine collapses. Caveat user.

jpeg\_add\_quant\_table (j\_compress\_ptr cinfo, int which\_tbl,

const unsigned int \*basic\_table,

int scale\_factor, boolean force\_baseline)

Allows an arbitrary quantization table to be created. which\_tbl

indicates which table slot to fill. basic\_table points to an array

of 64 unsigned ints given in normal array order. These values are

multiplied by scale\_factor/100 and then clamped to the range 1..65535

(or to 1..255 if force\_baseline is TRUE).

CAUTION: prior to library version 6a, jpeg\_add\_quant\_table expected

the basic table to be given in JPEG zigzag order. If you need to

write code that works with either older or newer versions of this

routine, you must check the library version number. Something like

"#if JPEG\_LIB\_VERSION >= 61" is the right test.

jpeg\_simple\_progression (j\_compress\_ptr cinfo)

Generates a default scan script for writing a progressive-JPEG file.

This is the recommended method of creating a progressive file,

unless you want to make a custom scan sequence. You must ensure that

the JPEG color space is set correctly before calling this routine.

Compression parameters (cinfo fields) include:

J\_DCT\_METHOD dct\_method

Selects the algorithm used for the DCT step. Choices are:

JDCT\_ISLOW: slow but accurate integer algorithm

JDCT\_IFAST: faster, less accurate integer method

JDCT\_FLOAT: floating-point method

JDCT\_DEFAULT: default method (normally JDCT\_ISLOW)

JDCT\_FASTEST: fastest method (normally JDCT\_IFAST)

The FLOAT method is very slightly more accurate than the ISLOW method,

but may give different results on different machines due to varying

roundoff behavior. The integer methods should give the same results

on all machines. On machines with sufficiently fast FP hardware, the

floating-point method may also be the fastest. The IFAST method is

considerably less accurate than the other two; its use is not

recommended if high quality is a concern. JDCT\_DEFAULT and

JDCT\_FASTEST are macros configurable by each installation.

J\_COLOR\_SPACE jpeg\_color\_space

int num\_components

The JPEG color space and corresponding number of components; see

"Special color spaces", below, for more info. We recommend using

jpeg\_set\_color\_space() if you want to change these.

boolean optimize\_coding

TRUE causes the compressor to compute optimal Huffman coding tables

for the image. This requires an extra pass over the data and

therefore costs a good deal of space and time. The default is

FALSE, which tells the compressor to use the supplied or default

Huffman tables. In most cases optimal tables save only a few percent

of file size compared to the default tables. Note that when this is

TRUE, you need not supply Huffman tables at all, and any you do

supply will be overwritten.

unsigned int restart\_interval

int restart\_in\_rows

To emit restart markers in the JPEG file, set one of these nonzero.

Set restart\_interval to specify the exact interval in MCU blocks.

Set restart\_in\_rows to specify the interval in MCU rows. (If

restart\_in\_rows is not 0, then restart\_interval is set after the

image width in MCUs is computed.) Defaults are zero (no restarts).

One restart marker per MCU row is often a good choice.

NOTE: the overhead of restart markers is higher in grayscale JPEG

files than in color files, and MUCH higher in progressive JPEGs.

If you use restarts, you may want to use larger intervals in those

cases.

const jpeg\_scan\_info \* scan\_info

int num\_scans

By default, scan\_info is NULL; this causes the compressor to write a

single-scan sequential JPEG file. If not NULL, scan\_info points to

an array of scan definition records of length num\_scans. The

compressor will then write a JPEG file having one scan for each scan

definition record. This is used to generate noninterleaved or

progressive JPEG files. The library checks that the scan array

defines a valid JPEG scan sequence. (jpeg\_simple\_progression creates

a suitable scan definition array for progressive JPEG.) This is

discussed further under "Progressive JPEG support".

int smoothing\_factor

If non-zero, the input image is smoothed; the value should be 1 for

minimal smoothing to 100 for maximum smoothing. Consult jcsample.c

for details of the smoothing algorithm. The default is zero.

boolean write\_JFIF\_header

If TRUE, a JFIF APP0 marker is emitted. jpeg\_set\_defaults() and

jpeg\_set\_colorspace() set this TRUE if a JFIF-legal JPEG color space

(ie, YCbCr or grayscale) is selected, otherwise FALSE.

UINT8 JFIF\_major\_version

UINT8 JFIF\_minor\_version

The version number to be written into the JFIF marker.

jpeg\_set\_defaults() initializes the version to 1.01 (major=minor=1).

You should set it to 1.02 (major=1, minor=2) if you plan to write

any JFIF 1.02 extension markers.

UINT8 density\_unit

UINT16 X\_density

UINT16 Y\_density

The resolution information to be written into the JFIF marker;

not used otherwise. density\_unit may be 0 for unknown,

1 for dots/inch, or 2 for dots/cm. The default values are 0,1,1

indicating square pixels of unknown size.

boolean write\_Adobe\_marker

If TRUE, an Adobe APP14 marker is emitted. jpeg\_set\_defaults() and

jpeg\_set\_colorspace() set this TRUE if JPEG color space RGB, CMYK,

or YCCK is selected, otherwise FALSE. It is generally a bad idea

to set both write\_JFIF\_header and write\_Adobe\_marker. In fact,

you probably shouldn't change the default settings at all --- the

default behavior ensures that the JPEG file's color space can be

recognized by the decoder.

JQUANT\_TBL \* quant\_tbl\_ptrs[NUM\_QUANT\_TBLS]

Pointers to coefficient quantization tables, one per table slot,

or NULL if no table is defined for a slot. Usually these should

be set via one of the above helper routines; jpeg\_add\_quant\_table()

is general enough to define any quantization table. The other

routines will set up table slot 0 for luminance quality and table

slot 1 for chrominance.

JHUFF\_TBL \* dc\_huff\_tbl\_ptrs[NUM\_HUFF\_TBLS]

JHUFF\_TBL \* ac\_huff\_tbl\_ptrs[NUM\_HUFF\_TBLS]

Pointers to Huffman coding tables, one per table slot, or NULL if

no table is defined for a slot. Slots 0 and 1 are filled with the

JPEG sample tables by jpeg\_set\_defaults(). If you need to allocate

more table structures, jpeg\_alloc\_huff\_table() may be used.

Note that optimal Huffman tables can be computed for an image

by setting optimize\_coding, as discussed above; there's seldom

any need to mess with providing your own Huffman tables.

There are some additional cinfo fields which are not documented here

because you currently can't change them; for example, you can't set

arith\_code TRUE because arithmetic coding is unsupported.

Per-component parameters are stored in the struct cinfo.comp\_info[i] for

component number i. Note that components here refer to components of the

JPEG color space, \*not\* the source image color space. A suitably large

comp\_info[] array is allocated by jpeg\_set\_defaults(); if you choose not

to use that routine, it's up to you to allocate the array.

int component\_id

The one-byte identifier code to be recorded in the JPEG file for

this component. For the standard color spaces, we recommend you

leave the default values alone.

int h\_samp\_factor

int v\_samp\_factor

Horizontal and vertical sampling factors for the component; must

be 1..4 according to the JPEG standard. Note that larger sampling

factors indicate a higher-resolution component; many people find

this behavior quite unintuitive. The default values are 2,2 for

luminance components and 1,1 for chrominance components, except

for grayscale where 1,1 is used.

int quant\_tbl\_no

Quantization table number for component. The default value is

0 for luminance components and 1 for chrominance components.

int dc\_tbl\_no

int ac\_tbl\_no

DC and AC entropy coding table numbers. The default values are

0 for luminance components and 1 for chrominance components.

int component\_index

Must equal the component's index in comp\_info[]. (Beginning in

release v6, the compressor library will fill this in automatically;

you don't have to.)

Decompression parameter selection

---------------------------------

Decompression parameter selection is somewhat simpler than compression

parameter selection, since all of the JPEG internal parameters are

recorded in the source file and need not be supplied by the application.

(Unless you are working with abbreviated files, in which case see

"Abbreviated datastreams", below.) Decompression parameters control

the postprocessing done on the image to deliver it in a format suitable

for the application's use. Many of the parameters control speed/quality

tradeoffs, in which faster decompression may be obtained at the price of

a poorer-quality image. The defaults select the highest quality (slowest)

processing.

The following fields in the JPEG object are set by jpeg\_read\_header() and

may be useful to the application in choosing decompression parameters:

JDIMENSION image\_width Width and height of image

JDIMENSION image\_height

int num\_components Number of color components

J\_COLOR\_SPACE jpeg\_color\_space Colorspace of image

boolean saw\_JFIF\_marker TRUE if a JFIF APP0 marker was seen

UINT8 JFIF\_major\_version Version information from JFIF marker

UINT8 JFIF\_minor\_version

UINT8 density\_unit Resolution data from JFIF marker

UINT16 X\_density

UINT16 Y\_density

boolean saw\_Adobe\_marker TRUE if an Adobe APP14 marker was seen

UINT8 Adobe\_transform Color transform code from Adobe marker

The JPEG color space, unfortunately, is something of a guess since the JPEG

standard proper does not provide a way to record it. In practice most files

adhere to the JFIF or Adobe conventions, and the decoder will recognize these

correctly. See "Special color spaces", below, for more info.

The decompression parameters that determine the basic properties of the

returned image are:

J\_COLOR\_SPACE out\_color\_space

Output color space. jpeg\_read\_header() sets an appropriate default

based on jpeg\_color\_space; typically it will be RGB or grayscale.

The application can change this field to request output in a different

colorspace. For example, set it to JCS\_GRAYSCALE to get grayscale

output from a color file. (This is useful for previewing: grayscale

output is faster than full color since the color components need not

be processed.) Note that not all possible color space transforms are

currently implemented; you may need to extend jdcolor.c if you want an

unusual conversion.

unsigned int scale\_num, scale\_denom

Scale the image by the fraction scale\_num/scale\_denom. Default is

1/1, or no scaling. Currently, the only supported scaling ratios

are 1/1, 1/2, 1/4, and 1/8. (The library design allows for arbitrary

scaling ratios but this is not likely to be implemented any time soon.)

Smaller scaling ratios permit significantly faster decoding since

fewer pixels need be processed and a simpler IDCT method can be used.

boolean quantize\_colors

If set TRUE, colormapped output will be delivered. Default is FALSE,

meaning that full-color output will be delivered.

The next three parameters are relevant only if quantize\_colors is TRUE.

int desired\_number\_of\_colors

Maximum number of colors to use in generating a library-supplied color

map (the actual number of colors is returned in a different field).

Default 256. Ignored when the application supplies its own color map.

boolean two\_pass\_quantize

If TRUE, an extra pass over the image is made to select a custom color

map for the image. This usually looks a lot better than the one-size-

fits-all colormap that is used otherwise. Default is TRUE. Ignored

when the application supplies its own color map.

J\_DITHER\_MODE dither\_mode

Selects color dithering method. Supported values are:

JDITHER\_NONE no dithering: fast, very low quality

JDITHER\_ORDERED ordered dither: moderate speed and quality

JDITHER\_FS Floyd-Steinberg dither: slow, high quality

Default is JDITHER\_FS. (At present, ordered dither is implemented

only in the single-pass, standard-colormap case. If you ask for

ordered dither when two\_pass\_quantize is TRUE or when you supply

an external color map, you'll get F-S dithering.)

When quantize\_colors is TRUE, the target color map is described by the next

two fields. colormap is set to NULL by jpeg\_read\_header(). The application

can supply a color map by setting colormap non-NULL and setting

actual\_number\_of\_colors to the map size. Otherwise, jpeg\_start\_decompress()

selects a suitable color map and sets these two fields itself.

[Implementation restriction: at present, an externally supplied colormap is

only accepted for 3-component output color spaces.]

JSAMPARRAY colormap

The color map, represented as a 2-D pixel array of out\_color\_components

rows and actual\_number\_of\_colors columns. Ignored if not quantizing.

CAUTION: if the JPEG library creates its own colormap, the storage

pointed to by this field is released by jpeg\_finish\_decompress().

Copy the colormap somewhere else first, if you want to save it.

int actual\_number\_of\_colors

The number of colors in the color map.

Additional decompression parameters that the application may set include:

J\_DCT\_METHOD dct\_method

Selects the algorithm used for the DCT step. Choices are the same

as described above for compression.

boolean do\_fancy\_upsampling

If TRUE, do careful upsampling of chroma components. If FALSE,

a faster but sloppier method is used. Default is TRUE. The visual

impact of the sloppier method is often very small.

boolean do\_block\_smoothing

If TRUE, interblock smoothing is applied in early stages of decoding

progressive JPEG files; if FALSE, not. Default is TRUE. Early

progression stages look "fuzzy" with smoothing, "blocky" without.

In any case, block smoothing ceases to be applied after the first few

AC coefficients are known to full accuracy, so it is relevant only

when using buffered-image mode for progressive images.

boolean enable\_1pass\_quant

boolean enable\_external\_quant

boolean enable\_2pass\_quant

These are significant only in buffered-image mode, which is

described in its own section below.

The output image dimensions are given by the following fields. These are

computed from the source image dimensions and the decompression parameters

by jpeg\_start\_decompress(). You can also call jpeg\_calc\_output\_dimensions()

to obtain the values that will result from the current parameter settings.

This can be useful if you are trying to pick a scaling ratio that will get

close to a desired target size. It's also important if you are using the

JPEG library's memory manager to allocate output buffer space, because you

are supposed to request such buffers \*before\* jpeg\_start\_decompress().

JDIMENSION output\_width Actual dimensions of output image.

JDIMENSION output\_height

int out\_color\_components Number of color components in out\_color\_space.

int output\_components Number of color components returned.

int rec\_outbuf\_height Recommended height of scanline buffer.

When quantizing colors, output\_components is 1, indicating a single color map

index per pixel. Otherwise it equals out\_color\_components. The output arrays

are required to be output\_width \* output\_components JSAMPLEs wide.

rec\_outbuf\_height is the recommended minimum height (in scanlines) of the

buffer passed to jpeg\_read\_scanlines(). If the buffer is smaller, the

library will still work, but time will be wasted due to unnecessary data

copying. In high-quality modes, rec\_outbuf\_height is always 1, but some

faster, lower-quality modes set it to larger values (typically 2 to 4).

If you are going to ask for a high-speed processing mode, you may as well

go to the trouble of honoring rec\_outbuf\_height so as to avoid data copying.

(An output buffer larger than rec\_outbuf\_height lines is OK, but won't

provide any material speed improvement over that height.)

Special color spaces

--------------------

The JPEG standard itself is "color blind" and doesn't specify any particular

color space. It is customary to convert color data to a luminance/chrominance

color space before compressing, since this permits greater compression. The

existing de-facto JPEG file format standards specify YCbCr or grayscale data

(JFIF), or grayscale, RGB, YCbCr, CMYK, or YCCK (Adobe). For special

applications such as multispectral images, other color spaces can be used,

but it must be understood that such files will be unportable.

The JPEG library can handle the most common colorspace conversions (namely

RGB <=> YCbCr and CMYK <=> YCCK). It can also deal with data of an unknown

color space, passing it through without conversion. If you deal extensively

with an unusual color space, you can easily extend the library to understand

additional color spaces and perform appropriate conversions.

For compression, the source data's color space is specified by field

in\_color\_space. This is transformed to the JPEG file's color space given

by jpeg\_color\_space. jpeg\_set\_defaults() chooses a reasonable JPEG color

space depending on in\_color\_space, but you can override this by calling

jpeg\_set\_colorspace(). Of course you must select a supported transformation.

jccolor.c currently supports the following transformations:

RGB => YCbCr

RGB => GRAYSCALE

YCbCr => GRAYSCALE

CMYK => YCCK

plus the null transforms: GRAYSCALE => GRAYSCALE, RGB => RGB,

YCbCr => YCbCr, CMYK => CMYK, YCCK => YCCK, and UNKNOWN => UNKNOWN.

The de-facto file format standards (JFIF and Adobe) specify APPn markers that

indicate the color space of the JPEG file. It is important to ensure that

these are written correctly, or omitted if the JPEG file's color space is not

one of the ones supported by the de-facto standards. jpeg\_set\_colorspace()

will set the compression parameters to include or omit the APPn markers

properly, so long as it is told the truth about the JPEG color space.

For example, if you are writing some random 3-component color space without

conversion, don't try to fake out the library by setting in\_color\_space and

jpeg\_color\_space to JCS\_YCbCr; use JCS\_UNKNOWN. You may want to write an

APPn marker of your own devising to identify the colorspace --- see "Special

markers", below.

When told that the color space is UNKNOWN, the library will default to using

luminance-quality compression parameters for all color components. You may

well want to change these parameters. See the source code for

jpeg\_set\_colorspace(), in jcparam.c, for details.

For decompression, the JPEG file's color space is given in jpeg\_color\_space,

and this is transformed to the output color space out\_color\_space.

jpeg\_read\_header's setting of jpeg\_color\_space can be relied on if the file

conforms to JFIF or Adobe conventions, but otherwise it is no better than a

guess. If you know the JPEG file's color space for certain, you can override

jpeg\_read\_header's guess by setting jpeg\_color\_space. jpeg\_read\_header also

selects a default output color space based on (its guess of) jpeg\_color\_space;

set out\_color\_space to override this. Again, you must select a supported

transformation. jdcolor.c currently supports

YCbCr => GRAYSCALE

YCbCr => RGB

GRAYSCALE => RGB

YCCK => CMYK

as well as the null transforms. (Since GRAYSCALE=>RGB is provided, an

application can force grayscale JPEGs to look like color JPEGs if it only

wants to handle one case.)

The two-pass color quantizer, jquant2.c, is specialized to handle RGB data

(it weights distances appropriately for RGB colors). You'll need to modify

the code if you want to use it for non-RGB output color spaces. Note that

jquant2.c is used to map to an application-supplied colormap as well as for

the normal two-pass colormap selection process.

CAUTION: it appears that Adobe Photoshop writes inverted data in CMYK JPEG

files: 0 represents 100% ink coverage, rather than 0% ink as you'd expect.

This is arguably a bug in Photoshop, but if you need to work with Photoshop

CMYK files, you will have to deal with it in your application. We cannot

"fix" this in the library by inverting the data during the CMYK<=>YCCK

transform, because that would break other applications, notably Ghostscript.

Photoshop versions prior to 3.0 write EPS files containing JPEG-encoded CMYK

data in the same inverted-YCCK representation used in bare JPEG files, but

the surrounding PostScript code performs an inversion using the PS image

operator. I am told that Photoshop 3.0 will write uninverted YCCK in

EPS/JPEG files, and will omit the PS-level inversion. (But the data

polarity used in bare JPEG files will not change in 3.0.) In either case,

the JPEG library must not invert the data itself, or else Ghostscript would

read these EPS files incorrectly.

Error handling

--------------

When the default error handler is used, any error detected inside the JPEG

routines will cause a message to be printed on stderr, followed by exit().

You can supply your own error handling routines to override this behavior

and to control the treatment of nonfatal warnings and trace/debug messages.

The file example.c illustrates the most common case, which is to have the

application regain control after an error rather than exiting.

The JPEG library never writes any message directly; it always goes through

the error handling routines. Three classes of messages are recognized:

\* Fatal errors: the library cannot continue.

\* Warnings: the library can continue, but the data is corrupt, and a

damaged output image is likely to result.

\* Trace/informational messages. These come with a trace level indicating

the importance of the message; you can control the verbosity of the

program by adjusting the maximum trace level that will be displayed.

You may, if you wish, simply replace the entire JPEG error handling module

(jerror.c) with your own code. However, you can avoid code duplication by

only replacing some of the routines depending on the behavior you need.

This is accomplished by calling jpeg\_std\_error() as usual, but then overriding

some of the method pointers in the jpeg\_error\_mgr struct, as illustrated by

example.c.

All of the error handling routines will receive a pointer to the JPEG object

(a j\_common\_ptr which points to either a jpeg\_compress\_struct or a

jpeg\_decompress\_struct; if you need to tell which, test the is\_decompressor

field). This struct includes a pointer to the error manager struct in its

"err" field. Frequently, custom error handler routines will need to access

additional data which is not known to the JPEG library or the standard error

handler. The most convenient way to do this is to embed either the JPEG

object or the jpeg\_error\_mgr struct in a larger structure that contains

additional fields; then casting the passed pointer provides access to the

additional fields. Again, see example.c for one way to do it. (Beginning

with IJG version 6b, there is also a void pointer "client\_data" in each

JPEG object, which the application can also use to find related data.

The library does not touch client\_data at all.)

The individual methods that you might wish to override are:

error\_exit (j\_common\_ptr cinfo)

Receives control for a fatal error. Information sufficient to

generate the error message has been stored in cinfo->err; call

output\_message to display it. Control must NOT return to the caller;

generally this routine will exit() or longjmp() somewhere.

Typically you would override this routine to get rid of the exit()

default behavior. Note that if you continue processing, you should

clean up the JPEG object with jpeg\_abort() or jpeg\_destroy().

output\_message (j\_common\_ptr cinfo)

Actual output of any JPEG message. Override this to send messages

somewhere other than stderr. Note that this method does not know

how to generate a message, only where to send it.

format\_message (j\_common\_ptr cinfo, char \* buffer)

Constructs a readable error message string based on the error info

stored in cinfo->err. This method is called by output\_message. Few

applications should need to override this method. One possible

reason for doing so is to implement dynamic switching of error message

language.

emit\_message (j\_common\_ptr cinfo, int msg\_level)

Decide whether or not to emit a warning or trace message; if so,

calls output\_message. The main reason for overriding this method

would be to abort on warnings. msg\_level is -1 for warnings,

0 and up for trace messages.

Only error\_exit() and emit\_message() are called from the rest of the JPEG

library; the other two are internal to the error handler.

The actual message texts are stored in an array of strings which is pointed to

by the field err->jpeg\_message\_table. The messages are numbered from 0 to

err->last\_jpeg\_message, and it is these code numbers that are used in the

JPEG library code. You could replace the message texts (for instance, with

messages in French or German) by changing the message table pointer. See

jerror.h for the default texts. CAUTION: this table will almost certainly

change or grow from one library version to the next.

It may be useful for an application to add its own message texts that are

handled by the same mechanism. The error handler supports a second "add-on"

message table for this purpose. To define an addon table, set the pointer

err->addon\_message\_table and the message numbers err->first\_addon\_message and

err->last\_addon\_message. If you number the addon messages beginning at 1000

or so, you won't have to worry about conflicts with the library's built-in

messages. See the sample applications cjpeg/djpeg for an example of using

addon messages (the addon messages are defined in cderror.h).

Actual invocation of the error handler is done via macros defined in jerror.h:

ERREXITn(...) for fatal errors

WARNMSn(...) for corrupt-data warnings

TRACEMSn(...) for trace and informational messages.

These macros store the message code and any additional parameters into the

error handler struct, then invoke the error\_exit() or emit\_message() method.

The variants of each macro are for varying numbers of additional parameters.

The additional parameters are inserted into the generated message using

standard printf() format codes.

See jerror.h and jerror.c for further details.

Compressed data handling (source and destination managers)

----------------------------------------------------------

The JPEG compression library sends its compressed data to a "destination

manager" module. The default destination manager just writes the data to a

stdio stream, but you can provide your own manager to do something else.

Similarly, the decompression library calls a "source manager" to obtain the

compressed data; you can provide your own source manager if you want the data

to come from somewhere other than a stdio stream.

In both cases, compressed data is processed a bufferload at a time: the

destination or source manager provides a work buffer, and the library invokes

the manager only when the buffer is filled or emptied. (You could define a

one-character buffer to force the manager to be invoked for each byte, but

that would be rather inefficient.) The buffer's size and location are

controlled by the manager, not by the library. For example, if you desired to

decompress a JPEG datastream that was all in memory, you could just make the

buffer pointer and length point to the original data in memory. Then the

buffer-reload procedure would be invoked only if the decompressor ran off the

end of the datastream, which would indicate an erroneous datastream.

The work buffer is defined as an array of datatype JOCTET, which is generally

"char" or "unsigned char". On a machine where char is not exactly 8 bits

wide, you must define JOCTET as a wider data type and then modify the data

source and destination modules to transcribe the work arrays into 8-bit units

on external storage.

A data destination manager struct contains a pointer and count defining the

next byte to write in the work buffer and the remaining free space:

JOCTET \* next\_output\_byte; /\* => next byte to write in buffer \*/

size\_t free\_in\_buffer; /\* # of byte spaces remaining in buffer \*/

The library increments the pointer and decrements the count until the buffer

is filled. The manager's empty\_output\_buffer method must reset the pointer

and count. The manager is expected to remember the buffer's starting address

and total size in private fields not visible to the library.

A data destination manager provides three methods:

init\_destination (j\_compress\_ptr cinfo)

Initialize destination. This is called by jpeg\_start\_compress()

before any data is actually written. It must initialize

next\_output\_byte and free\_in\_buffer. free\_in\_buffer must be

initialized to a positive value.

empty\_output\_buffer (j\_compress\_ptr cinfo)

This is called whenever the buffer has filled (free\_in\_buffer

reaches zero). In typical applications, it should write out the

\*entire\* buffer (use the saved start address and buffer length;

ignore the current state of next\_output\_byte and free\_in\_buffer).

Then reset the pointer & count to the start of the buffer, and

return TRUE indicating that the buffer has been dumped.

free\_in\_buffer must be set to a positive value when TRUE is

returned. A FALSE return should only be used when I/O suspension is

desired (this operating mode is discussed in the next section).

term\_destination (j\_compress\_ptr cinfo)

Terminate destination --- called by jpeg\_finish\_compress() after all

data has been written. In most applications, this must flush any

data remaining in the buffer. Use either next\_output\_byte or

free\_in\_buffer to determine how much data is in the buffer.

term\_destination() is NOT called by jpeg\_abort() or jpeg\_destroy(). If you

want the destination manager to be cleaned up during an abort, you must do it

yourself.

You will also need code to create a jpeg\_destination\_mgr struct, fill in its

method pointers, and insert a pointer to the struct into the "dest" field of

the JPEG compression object. This can be done in-line in your setup code if

you like, but it's probably cleaner to provide a separate routine similar to

the jpeg\_stdio\_dest() routine of the supplied destination manager.

Decompression source managers follow a parallel design, but with some

additional frammishes. The source manager struct contains a pointer and count

defining the next byte to read from the work buffer and the number of bytes

remaining:

const JOCTET \* next\_input\_byte; /\* => next byte to read from buffer \*/

size\_t bytes\_in\_buffer; /\* # of bytes remaining in buffer \*/

The library increments the pointer and decrements the count until the buffer

is emptied. The manager's fill\_input\_buffer method must reset the pointer and

count. In most applications, the manager must remember the buffer's starting

address and total size in private fields not visible to the library.

A data source manager provides five methods:

init\_source (j\_decompress\_ptr cinfo)

Initialize source. This is called by jpeg\_read\_header() before any

data is actually read. Unlike init\_destination(), it may leave

bytes\_in\_buffer set to 0 (in which case a fill\_input\_buffer() call

will occur immediately).

fill\_input\_buffer (j\_decompress\_ptr cinfo)

This is called whenever bytes\_in\_buffer has reached zero and more

data is wanted. In typical applications, it should read fresh data

into the buffer (ignoring the current state of next\_input\_byte and

bytes\_in\_buffer), reset the pointer & count to the start of the

buffer, and return TRUE indicating that the buffer has been reloaded.

It is not necessary to fill the buffer entirely, only to obtain at

least one more byte. bytes\_in\_buffer MUST be set to a positive value

if TRUE is returned. A FALSE return should only be used when I/O

suspension is desired (this mode is discussed in the next section).

skip\_input\_data (j\_decompress\_ptr cinfo, long num\_bytes)

Skip num\_bytes worth of data. The buffer pointer and count should

be advanced over num\_bytes input bytes, refilling the buffer as

needed. This is used to skip over a potentially large amount of

uninteresting data (such as an APPn marker). In some applications

it may be possible to optimize away the reading of the skipped data,

but it's not clear that being smart is worth much trouble; large

skips are uncommon. bytes\_in\_buffer may be zero on return.

A zero or negative skip count should be treated as a no-op.

resync\_to\_restart (j\_decompress\_ptr cinfo, int desired)

This routine is called only when the decompressor has failed to find

a restart (RSTn) marker where one is expected. Its mission is to

find a suitable point for resuming decompression. For most

applications, we recommend that you just use the default resync

procedure, jpeg\_resync\_to\_restart(). However, if you are able to back

up in the input data stream, or if you have a-priori knowledge about

the likely location of restart markers, you may be able to do better.

Read the read\_restart\_marker() and jpeg\_resync\_to\_restart() routines

in jdmarker.c if you think you'd like to implement your own resync

procedure.

term\_source (j\_decompress\_ptr cinfo)

Terminate source --- called by jpeg\_finish\_decompress() after all

data has been read. Often a no-op.

For both fill\_input\_buffer() and skip\_input\_data(), there is no such thing

as an EOF return. If the end of the file has been reached, the routine has

a choice of exiting via ERREXIT() or inserting fake data into the buffer.

In most cases, generating a warning message and inserting a fake EOI marker

is the best course of action --- this will allow the decompressor to output

however much of the image is there. In pathological cases, the decompressor

may swallow the EOI and again demand data ... just keep feeding it fake EOIs.

jdatasrc.c illustrates the recommended error recovery behavior.

term\_source() is NOT called by jpeg\_abort() or jpeg\_destroy(). If you want

the source manager to be cleaned up during an abort, you must do it yourself.

You will also need code to create a jpeg\_source\_mgr struct, fill in its method

pointers, and insert a pointer to the struct into the "src" field of the JPEG

decompression object. This can be done in-line in your setup code if you

like, but it's probably cleaner to provide a separate routine similar to the

jpeg\_stdio\_src() routine of the supplied source manager.

For more information, consult the stdio source and destination managers

in jdatasrc.c and jdatadst.c.

I/O suspension

--------------

Some applications need to use the JPEG library as an incremental memory-to-

memory filter: when the compressed data buffer is filled or emptied, they want

control to return to the outer loop, rather than expecting that the buffer can

be emptied or reloaded within the data source/destination manager subroutine.

The library supports this need by providing an "I/O suspension" mode, which we

describe in this section.

The I/O suspension mode is not a panacea: nothing is guaranteed about the

maximum amount of time spent in any one call to the library, so it will not

eliminate response-time problems in single-threaded applications. If you

need guaranteed response time, we suggest you "bite the bullet" and implement

a real multi-tasking capability.

To use I/O suspension, cooperation is needed between the calling application

and the data source or destination manager; you will always need a custom

source/destination manager. (Please read the previous section if you haven't

already.) The basic idea is that the empty\_output\_buffer() or

fill\_input\_buffer() routine is a no-op, merely returning FALSE to indicate

that it has done nothing. Upon seeing this, the JPEG library suspends

operation and returns to its caller. The surrounding application is

responsible for emptying or refilling the work buffer before calling the

JPEG library again.

Compression suspension:

For compression suspension, use an empty\_output\_buffer() routine that returns

FALSE; typically it will not do anything else. This will cause the

compressor to return to the caller of jpeg\_write\_scanlines(), with the return

value indicating that not all the supplied scanlines have been accepted.

The application must make more room in the output buffer, adjust the output

buffer pointer/count appropriately, and then call jpeg\_write\_scanlines()

again, pointing to the first unconsumed scanline.

When forced to suspend, the compressor will backtrack to a convenient stopping

point (usually the start of the current MCU); it will regenerate some output

data when restarted. Therefore, although empty\_output\_buffer() is only

called when the buffer is filled, you should NOT write out the entire buffer

after a suspension. Write only the data up to the current position of

next\_output\_byte/free\_in\_buffer. The data beyond that point will be

regenerated after resumption.

Because of the backtracking behavior, a good-size output buffer is essential

for efficiency; you don't want the compressor to suspend often. (In fact, an

overly small buffer could lead to infinite looping, if a single MCU required

more data than would fit in the buffer.) We recommend a buffer of at least

several Kbytes. You may want to insert explicit code to ensure that you don't

call jpeg\_write\_scanlines() unless there is a reasonable amount of space in

the output buffer; in other words, flush the buffer before trying to compress

more data.

The compressor does not allow suspension while it is trying to write JPEG

markers at the beginning and end of the file. This means that:

\* At the beginning of a compression operation, there must be enough free

space in the output buffer to hold the header markers (typically 600 or

so bytes). The recommended buffer size is bigger than this anyway, so

this is not a problem as long as you start with an empty buffer. However,

this restriction might catch you if you insert large special markers, such

as a JFIF thumbnail image, without flushing the buffer afterwards.

\* When you call jpeg\_finish\_compress(), there must be enough space in the

output buffer to emit any buffered data and the final EOI marker. In the

current implementation, half a dozen bytes should suffice for this, but

for safety's sake we recommend ensuring that at least 100 bytes are free

before calling jpeg\_finish\_compress().

A more significant restriction is that jpeg\_finish\_compress() cannot suspend.

This means you cannot use suspension with multi-pass operating modes, namely

Huffman code optimization and multiple-scan output. Those modes write the

whole file during jpeg\_finish\_compress(), which will certainly result in

buffer overrun. (Note that this restriction applies only to compression,

not decompression. The decompressor supports input suspension in all of its

operating modes.)

Decompression suspension:

For decompression suspension, use a fill\_input\_buffer() routine that simply

returns FALSE (except perhaps during error recovery, as discussed below).

This will cause the decompressor to return to its caller with an indication

that suspension has occurred. This can happen at four places:

\* jpeg\_read\_header(): will return JPEG\_SUSPENDED.

\* jpeg\_start\_decompress(): will return FALSE, rather than its usual TRUE.

\* jpeg\_read\_scanlines(): will return the number of scanlines already

completed (possibly 0).

\* jpeg\_finish\_decompress(): will return FALSE, rather than its usual TRUE.

The surrounding application must recognize these cases, load more data into

the input buffer, and repeat the call. In the case of jpeg\_read\_scanlines(),

increment the passed pointers past any scanlines successfully read.

Just as with compression, the decompressor will typically backtrack to a

convenient restart point before suspending. When fill\_input\_buffer() is

called, next\_input\_byte/bytes\_in\_buffer point to the current restart point,

which is where the decompressor will backtrack to if FALSE is returned.

The data beyond that position must NOT be discarded if you suspend; it needs

to be re-read upon resumption. In most implementations, you'll need to shift

this data down to the start of your work buffer and then load more data after

it. Again, this behavior means that a several-Kbyte work buffer is essential

for decent performance; furthermore, you should load a reasonable amount of

new data before resuming decompression. (If you loaded, say, only one new

byte each time around, you could waste a LOT of cycles.)

The skip\_input\_data() source manager routine requires special care in a

suspension scenario. This routine is NOT granted the ability to suspend the

decompressor; it can decrement bytes\_in\_buffer to zero, but no more. If the

requested skip distance exceeds the amount of data currently in the input

buffer, then skip\_input\_data() must set bytes\_in\_buffer to zero and record the

additional skip distance somewhere else. The decompressor will immediately

call fill\_input\_buffer(), which should return FALSE, which will cause a

suspension return. The surrounding application must then arrange to discard

the recorded number of bytes before it resumes loading the input buffer.

(Yes, this design is rather baroque, but it avoids complexity in the far more

common case where a non-suspending source manager is used.)

If the input data has been exhausted, we recommend that you emit a warning

and insert dummy EOI markers just as a non-suspending data source manager

would do. This can be handled either in the surrounding application logic or

within fill\_input\_buffer(); the latter is probably more efficient. If

fill\_input\_buffer() knows that no more data is available, it can set the

pointer/count to point to a dummy EOI marker and then return TRUE just as

though it had read more data in a non-suspending situation.

The decompressor does not attempt to suspend within standard JPEG markers;

instead it will backtrack to the start of the marker and reprocess the whole

marker next time. Hence the input buffer must be large enough to hold the

longest standard marker in the file. Standard JPEG markers should normally

not exceed a few hundred bytes each (DHT tables are typically the longest).

We recommend at least a 2K buffer for performance reasons, which is much

larger than any correct marker is likely to be. For robustness against

damaged marker length counts, you may wish to insert a test in your

application for the case that the input buffer is completely full and yet

the decoder has suspended without consuming any data --- otherwise, if this

situation did occur, it would lead to an endless loop. (The library can't

provide this test since it has no idea whether "the buffer is full", or

even whether there is a fixed-size input buffer.)

The input buffer would need to be 64K to allow for arbitrary COM or APPn

markers, but these are handled specially: they are either saved into allocated

memory, or skipped over by calling skip\_input\_data(). In the former case,

suspension is handled correctly, and in the latter case, the problem of

buffer overrun is placed on skip\_input\_data's shoulders, as explained above.

Note that if you provide your own marker handling routine for large markers,

you should consider how to deal with buffer overflow.

Multiple-buffer management:

In some applications it is desirable to store the compressed data in a linked

list of buffer areas, so as to avoid data copying. This can be handled by

having empty\_output\_buffer() or fill\_input\_buffer() set the pointer and count

to reference the next available buffer; FALSE is returned only if no more

buffers are available. Although seemingly straightforward, there is a

pitfall in this approach: the backtrack that occurs when FALSE is returned

could back up into an earlier buffer. For example, when fill\_input\_buffer()

is called, the current pointer & count indicate the backtrack restart point.

Since fill\_input\_buffer() will set the pointer and count to refer to a new

buffer, the restart position must be saved somewhere else. Suppose a second

call to fill\_input\_buffer() occurs in the same library call, and no

additional input data is available, so fill\_input\_buffer must return FALSE.

If the JPEG library has not moved the pointer/count forward in the current

buffer, then \*the correct restart point is the saved position in the prior

buffer\*. Prior buffers may be discarded only after the library establishes

a restart point within a later buffer. Similar remarks apply for output into

a chain of buffers.

The library will never attempt to backtrack over a skip\_input\_data() call,

so any skipped data can be permanently discarded. You still have to deal

with the case of skipping not-yet-received data, however.

It's much simpler to use only a single buffer; when fill\_input\_buffer() is

called, move any unconsumed data (beyond the current pointer/count) down to

the beginning of this buffer and then load new data into the remaining buffer

space. This approach requires a little more data copying but is far easier

to get right.

Progressive JPEG support

------------------------

Progressive JPEG rearranges the stored data into a series of scans of

increasing quality. In situations where a JPEG file is transmitted across a

slow communications link, a decoder can generate a low-quality image very

quickly from the first scan, then gradually improve the displayed quality as

more scans are received. The final image after all scans are complete is

identical to that of a regular (sequential) JPEG file of the same quality

setting. Progressive JPEG files are often slightly smaller than equivalent

sequential JPEG files, but the possibility of incremental display is the main

reason for using progressive JPEG.

The IJG encoder library generates progressive JPEG files when given a

suitable "scan script" defining how to divide the data into scans.

Creation of progressive JPEG files is otherwise transparent to the encoder.

Progressive JPEG files can also be read transparently by the decoder library.

If the decoding application simply uses the library as defined above, it

will receive a final decoded image without any indication that the file was

progressive. Of course, this approach does not allow incremental display.

To perform incremental display, an application needs to use the decoder

library's "buffered-image" mode, in which it receives a decoded image

multiple times.

Each displayed scan requires about as much work to decode as a full JPEG

image of the same size, so the decoder must be fairly fast in relation to the

data transmission rate in order to make incremental display useful. However,

it is possible to skip displaying the image and simply add the incoming bits

to the decoder's coefficient buffer. This is fast because only Huffman

decoding need be done, not IDCT, upsampling, colorspace conversion, etc.

The IJG decoder library allows the application to switch dynamically between

displaying the image and simply absorbing the incoming bits. A properly

coded application can automatically adapt the number of display passes to

suit the time available as the image is received. Also, a final

higher-quality display cycle can be performed from the buffered data after

the end of the file is reached.

Progressive compression:

To create a progressive JPEG file (or a multiple-scan sequential JPEG file),

set the scan\_info cinfo field to point to an array of scan descriptors, and

perform compression as usual. Instead of constructing your own scan list,

you can call the jpeg\_simple\_progression() helper routine to create a

recommended progression sequence; this method should be used by all

applications that don't want to get involved in the nitty-gritty of

progressive scan sequence design. (If you want to provide user control of

scan sequences, you may wish to borrow the scan script reading code found

in rdswitch.c, so that you can read scan script files just like cjpeg's.)

When scan\_info is not NULL, the compression library will store DCT'd data

into a buffer array as jpeg\_write\_scanlines() is called, and will emit all

the requested scans during jpeg\_finish\_compress(). This implies that

multiple-scan output cannot be created with a suspending data destination

manager, since jpeg\_finish\_compress() does not support suspension. We

should also note that the compressor currently forces Huffman optimization

mode when creating a progressive JPEG file, because the default Huffman

tables are unsuitable for progressive files.

Progressive decompression:

When buffered-image mode is not used, the decoder library will read all of

a multi-scan file during jpeg\_start\_decompress(), so that it can provide a

final decoded image. (Here "multi-scan" means either progressive or

multi-scan sequential.) This makes multi-scan files transparent to the

decoding application. However, existing applications that used suspending

input with version 5 of the IJG library will need to be modified to check

for a suspension return from jpeg\_start\_decompress().

To perform incremental display, an application must use the library's

buffered-image mode. This is described in the next section.

Buffered-image mode

-------------------

In buffered-image mode, the library stores the partially decoded image in a

coefficient buffer, from which it can be read out as many times as desired.

This mode is typically used for incremental display of progressive JPEG files,

but it can be used with any JPEG file. Each scan of a progressive JPEG file

adds more data (more detail) to the buffered image. The application can

display in lockstep with the source file (one display pass per input scan),

or it can allow input processing to outrun display processing. By making

input and display processing run independently, it is possible for the

application to adapt progressive display to a wide range of data transmission

rates.

The basic control flow for buffered-image decoding is

jpeg\_create\_decompress()

set data source

jpeg\_read\_header()

set overall decompression parameters

cinfo.buffered\_image = TRUE; /\* select buffered-image mode \*/

jpeg\_start\_decompress()

for (each output pass) {

adjust output decompression parameters if required

jpeg\_start\_output() /\* start a new output pass \*/

for (all scanlines in image) {

jpeg\_read\_scanlines()

display scanlines

}

jpeg\_finish\_output() /\* terminate output pass \*/

}

jpeg\_finish\_decompress()

jpeg\_destroy\_decompress()

This differs from ordinary unbuffered decoding in that there is an additional

level of looping. The application can choose how many output passes to make

and how to display each pass.

The simplest approach to displaying progressive images is to do one display

pass for each scan appearing in the input file. In this case the outer loop

condition is typically

while (! jpeg\_input\_complete(&cinfo))

and the start-output call should read

jpeg\_start\_output(&cinfo, cinfo.input\_scan\_number);

The second parameter to jpeg\_start\_output() indicates which scan of the input

file is to be displayed; the scans are numbered starting at 1 for this

purpose. (You can use a loop counter starting at 1 if you like, but using

the library's input scan counter is easier.) The library automatically reads

data as necessary to complete each requested scan, and jpeg\_finish\_output()

advances to the next scan or end-of-image marker (hence input\_scan\_number

will be incremented by the time control arrives back at jpeg\_start\_output()).

With this technique, data is read from the input file only as needed, and

input and output processing run in lockstep.

After reading the final scan and reaching the end of the input file, the

buffered image remains available; it can be read additional times by

repeating the jpeg\_start\_output()/jpeg\_read\_scanlines()/jpeg\_finish\_output()

sequence. For example, a useful technique is to use fast one-pass color

quantization for display passes made while the image is arriving, followed by

a final display pass using two-pass quantization for highest quality. This

is done by changing the library parameters before the final output pass.

Changing parameters between passes is discussed in detail below.

In general the last scan of a progressive file cannot be recognized as such

until after it is read, so a post-input display pass is the best approach if

you want special processing in the final pass.

When done with the image, be sure to call jpeg\_finish\_decompress() to release

the buffered image (or just use jpeg\_destroy\_decompress()).

If input data arrives faster than it can be displayed, the application can

cause the library to decode input data in advance of what's needed to produce

output. This is done by calling the routine jpeg\_consume\_input().

The return value is one of the following:

JPEG\_REACHED\_SOS: reached an SOS marker (the start of a new scan)

JPEG\_REACHED\_EOI: reached the EOI marker (end of image)

JPEG\_ROW\_COMPLETED: completed reading one MCU row of compressed data

JPEG\_SCAN\_COMPLETED: completed reading last MCU row of current scan

JPEG\_SUSPENDED: suspended before completing any of the above

(JPEG\_SUSPENDED can occur only if a suspending data source is used.) This

routine can be called at any time after initializing the JPEG object. It

reads some additional data and returns when one of the indicated significant

events occurs. (If called after the EOI marker is reached, it will

immediately return JPEG\_REACHED\_EOI without attempting to read more data.)

The library's output processing will automatically call jpeg\_consume\_input()

whenever the output processing overtakes the input; thus, simple lockstep

display requires no direct calls to jpeg\_consume\_input(). But by adding

calls to jpeg\_consume\_input(), you can absorb data in advance of what is

being displayed. This has two benefits:

\* You can limit buildup of unprocessed data in your input buffer.

\* You can eliminate extra display passes by paying attention to the

state of the library's input processing.

The first of these benefits only requires interspersing calls to

jpeg\_consume\_input() with your display operations and any other processing

you may be doing. To avoid wasting cycles due to backtracking, it's best to

call jpeg\_consume\_input() only after a hundred or so new bytes have arrived.

This is discussed further under "I/O suspension", above. (Note: the JPEG

library currently is not thread-safe. You must not call jpeg\_consume\_input()

from one thread of control if a different library routine is working on the

same JPEG object in another thread.)

When input arrives fast enough that more than one new scan is available

before you start a new output pass, you may as well skip the output pass

corresponding to the completed scan. This occurs for free if you pass

cinfo.input\_scan\_number as the target scan number to jpeg\_start\_output().

The input\_scan\_number field is simply the index of the scan currently being

consumed by the input processor. You can ensure that this is up-to-date by

emptying the input buffer just before calling jpeg\_start\_output(): call

jpeg\_consume\_input() repeatedly until it returns JPEG\_SUSPENDED or

JPEG\_REACHED\_EOI.

The target scan number passed to jpeg\_start\_output() is saved in the

cinfo.output\_scan\_number field. The library's output processing calls

jpeg\_consume\_input() whenever the current input scan number and row within

that scan is less than or equal to the current output scan number and row.

Thus, input processing can "get ahead" of the output processing but is not

allowed to "fall behind". You can achieve several different effects by

manipulating this interlock rule. For example, if you pass a target scan

number greater than the current input scan number, the output processor will

wait until that scan starts to arrive before producing any output. (To avoid

an infinite loop, the target scan number is automatically reset to the last

scan number when the end of image is reached. Thus, if you specify a large

target scan number, the library will just absorb the entire input file and

then perform an output pass. This is effectively the same as what

jpeg\_start\_decompress() does when you don't select buffered-image mode.)

When you pass a target scan number equal to the current input scan number,

the image is displayed no faster than the current input scan arrives. The

final possibility is to pass a target scan number less than the current input

scan number; this disables the input/output interlock and causes the output

processor to simply display whatever it finds in the image buffer, without

waiting for input. (However, the library will not accept a target scan

number less than one, so you can't avoid waiting for the first scan.)

When data is arriving faster than the output display processing can advance

through the image, jpeg\_consume\_input() will store data into the buffered

image beyond the point at which the output processing is reading data out

again. If the input arrives fast enough, it may "wrap around" the buffer to

the point where the input is more than one whole scan ahead of the output.

If the output processing simply proceeds through its display pass without

paying attention to the input, the effect seen on-screen is that the lower

part of the image is one or more scans better in quality than the upper part.

Then, when the next output scan is started, you have a choice of what target

scan number to use. The recommended choice is to use the current input scan

number at that time, which implies that you've skipped the output scans

corresponding to the input scans that were completed while you processed the

previous output scan. In this way, the decoder automatically adapts its

speed to the arriving data, by skipping output scans as necessary to keep up

with the arriving data.

When using this strategy, you'll want to be sure that you perform a final

output pass after receiving all the data; otherwise your last display may not

be full quality across the whole screen. So the right outer loop logic is

something like this:

do {

absorb any waiting input by calling jpeg\_consume\_input()

final\_pass = jpeg\_input\_complete(&cinfo);

adjust output decompression parameters if required

jpeg\_start\_output(&cinfo, cinfo.input\_scan\_number);

...

jpeg\_finish\_output()

} while (! final\_pass);

rather than quitting as soon as jpeg\_input\_complete() returns TRUE. This

arrangement makes it simple to use higher-quality decoding parameters

for the final pass. But if you don't want to use special parameters for

the final pass, the right loop logic is like this:

for (;;) {

absorb any waiting input by calling jpeg\_consume\_input()

jpeg\_start\_output(&cinfo, cinfo.input\_scan\_number);

...

jpeg\_finish\_output()

if (jpeg\_input\_complete(&cinfo) &&

cinfo.input\_scan\_number == cinfo.output\_scan\_number)

break;

}

In this case you don't need to know in advance whether an output pass is to

be the last one, so it's not necessary to have reached EOF before starting

the final output pass; rather, what you want to test is whether the output

pass was performed in sync with the final input scan. This form of the loop

will avoid an extra output pass whenever the decoder is able (or nearly able)

to keep up with the incoming data.

When the data transmission speed is high, you might begin a display pass,

then find that much or all of the file has arrived before you can complete

the pass. (You can detect this by noting the JPEG\_REACHED\_EOI return code

from jpeg\_consume\_input(), or equivalently by testing jpeg\_input\_complete().)

In this situation you may wish to abort the current display pass and start a

new one using the newly arrived information. To do so, just call

jpeg\_finish\_output() and then start a new pass with jpeg\_start\_output().

A variant strategy is to abort and restart display if more than one complete

scan arrives during an output pass; this can be detected by noting

JPEG\_REACHED\_SOS returns and/or examining cinfo.input\_scan\_number. This

idea should be employed with caution, however, since the display process

might never get to the bottom of the image before being aborted, resulting

in the lower part of the screen being several passes worse than the upper.

In most cases it's probably best to abort an output pass only if the whole

file has arrived and you want to begin the final output pass immediately.

When receiving data across a communication link, we recommend always using

the current input scan number for the output target scan number; if a

higher-quality final pass is to be done, it should be started (aborting any

incomplete output pass) as soon as the end of file is received. However,

many other strategies are possible. For example, the application can examine

the parameters of the current input scan and decide whether to display it or

not. If the scan contains only chroma data, one might choose not to use it

as the target scan, expecting that the scan will be small and will arrive

quickly. To skip to the next scan, call jpeg\_consume\_input() until it

returns JPEG\_REACHED\_SOS or JPEG\_REACHED\_EOI. Or just use the next higher

number as the target scan for jpeg\_start\_output(); but that method doesn't

let you inspect the next scan's parameters before deciding to display it.

In buffered-image mode, jpeg\_start\_decompress() never performs input and

thus never suspends. An application that uses input suspension with

buffered-image mode must be prepared for suspension returns from these

routines:

\* jpeg\_start\_output() performs input only if you request 2-pass quantization

and the target scan isn't fully read yet. (This is discussed below.)

\* jpeg\_read\_scanlines(), as always, returns the number of scanlines that it

was able to produce before suspending.

\* jpeg\_finish\_output() will read any markers following the target scan,

up to the end of the file or the SOS marker that begins another scan.

(But it reads no input if jpeg\_consume\_input() has already reached the

end of the file or a SOS marker beyond the target output scan.)

\* jpeg\_finish\_decompress() will read until the end of file, and thus can

suspend if the end hasn't already been reached (as can be tested by

calling jpeg\_input\_complete()).

jpeg\_start\_output(), jpeg\_finish\_output(), and jpeg\_finish\_decompress()

all return TRUE if they completed their tasks, FALSE if they had to suspend.

In the event of a FALSE return, the application must load more input data

and repeat the call. Applications that use non-suspending data sources need

not check the return values of these three routines.

It is possible to change decoding parameters between output passes in the

buffered-image mode. The decoder library currently supports only very

limited changes of parameters. ONLY THE FOLLOWING parameter changes are

allowed after jpeg\_start\_decompress() is called:

\* dct\_method can be changed before each call to jpeg\_start\_output().

For example, one could use a fast DCT method for early scans, changing

to a higher quality method for the final scan.

\* dither\_mode can be changed before each call to jpeg\_start\_output();

of course this has no impact if not using color quantization. Typically

one would use ordered dither for initial passes, then switch to

Floyd-Steinberg dither for the final pass. Caution: changing dither mode

can cause more memory to be allocated by the library. Although the amount

of memory involved is not large (a scanline or so), it may cause the

initial max\_memory\_to\_use specification to be exceeded, which in the worst

case would result in an out-of-memory failure.

\* do\_block\_smoothing can be changed before each call to jpeg\_start\_output().

This setting is relevant only when decoding a progressive JPEG image.

During the first DC-only scan, block smoothing provides a very "fuzzy" look

instead of the very "blocky" look seen without it; which is better seems a

matter of personal taste. But block smoothing is nearly always a win

during later stages, especially when decoding a successive-approximation

image: smoothing helps to hide the slight blockiness that otherwise shows

up on smooth gradients until the lowest coefficient bits are sent.

\* Color quantization mode can be changed under the rules described below.

You \*cannot\* change between full-color and quantized output (because that

would alter the required I/O buffer sizes), but you can change which

quantization method is used.

When generating color-quantized output, changing quantization method is a

very useful way of switching between high-speed and high-quality display.

The library allows you to change among its three quantization methods:

1. Single-pass quantization to a fixed color cube.

Selected by cinfo.two\_pass\_quantize = FALSE and cinfo.colormap = NULL.

2. Single-pass quantization to an application-supplied colormap.

Selected by setting cinfo.colormap to point to the colormap (the value of

two\_pass\_quantize is ignored); also set cinfo.actual\_number\_of\_colors.

3. Two-pass quantization to a colormap chosen specifically for the image.

Selected by cinfo.two\_pass\_quantize = TRUE and cinfo.colormap = NULL.

(This is the default setting selected by jpeg\_read\_header, but it is

probably NOT what you want for the first pass of progressive display!)

These methods offer successively better quality and lesser speed. However,

only the first method is available for quantizing in non-RGB color spaces.

IMPORTANT: because the different quantizer methods have very different

working-storage requirements, the library requires you to indicate which

one(s) you intend to use before you call jpeg\_start\_decompress(). (If we did

not require this, the max\_memory\_to\_use setting would be a complete fiction.)

You do this by setting one or more of these three cinfo fields to TRUE:

enable\_1pass\_quant Fixed color cube colormap

enable\_external\_quant Externally-supplied colormap

enable\_2pass\_quant Two-pass custom colormap

All three are initialized FALSE by jpeg\_read\_header(). But

jpeg\_start\_decompress() automatically sets TRUE the one selected by the

current two\_pass\_quantize and colormap settings, so you only need to set the

enable flags for any other quantization methods you plan to change to later.

After setting the enable flags correctly at jpeg\_start\_decompress() time, you

can change to any enabled quantization method by setting two\_pass\_quantize

and colormap properly just before calling jpeg\_start\_output(). The following

special rules apply:

1. You must explicitly set cinfo.colormap to NULL when switching to 1-pass

or 2-pass mode from a different mode, or when you want the 2-pass

quantizer to be re-run to generate a new colormap.

2. To switch to an external colormap, or to change to a different external

colormap than was used on the prior pass, you must call

jpeg\_new\_colormap() after setting cinfo.colormap.

NOTE: if you want to use the same colormap as was used in the prior pass,

you should not do either of these things. This will save some nontrivial

switchover costs.

(These requirements exist because cinfo.colormap will always be non-NULL

after completing a prior output pass, since both the 1-pass and 2-pass

quantizers set it to point to their output colormaps. Thus you have to

do one of these two things to notify the library that something has changed.

Yup, it's a bit klugy, but it's necessary to do it this way for backwards

compatibility.)

Note that in buffered-image mode, the library generates any requested colormap

during jpeg\_start\_output(), not during jpeg\_start\_decompress().

When using two-pass quantization, jpeg\_start\_output() makes a pass over the

buffered image to determine the optimum color map; it therefore may take a

significant amount of time, whereas ordinarily it does little work. The

progress monitor hook is called during this pass, if defined. It is also

important to realize that if the specified target scan number is greater than

or equal to the current input scan number, jpeg\_start\_output() will attempt

to consume input as it makes this pass. If you use a suspending data source,

you need to check for a FALSE return from jpeg\_start\_output() under these

conditions. The combination of 2-pass quantization and a not-yet-fully-read

target scan is the only case in which jpeg\_start\_output() will consume input.

Application authors who support buffered-image mode may be tempted to use it

for all JPEG images, even single-scan ones. This will work, but it is

inefficient: there is no need to create an image-sized coefficient buffer for

single-scan images. Requesting buffered-image mode for such an image wastes

memory. Worse, it can cost time on large images, since the buffered data has

to be swapped out or written to a temporary file. If you are concerned about

maximum performance on baseline JPEG files, you should use buffered-image

mode only when the incoming file actually has multiple scans. This can be

tested by calling jpeg\_has\_multiple\_scans(), which will return a correct

result at any time after jpeg\_read\_header() completes.

It is also worth noting that when you use jpeg\_consume\_input() to let input

processing get ahead of output processing, the resulting pattern of access to

the coefficient buffer is quite nonsequential. It's best to use the memory

manager jmemnobs.c if you can (ie, if you have enough real or virtual main

memory). If not, at least make sure that max\_memory\_to\_use is set as high as

possible. If the JPEG memory manager has to use a temporary file, you will

probably see a lot of disk traffic and poor performance. (This could be

improved with additional work on the memory manager, but we haven't gotten

around to it yet.)

In some applications it may be convenient to use jpeg\_consume\_input() for all

input processing, including reading the initial markers; that is, you may

wish to call jpeg\_consume\_input() instead of jpeg\_read\_header() during

startup. This works, but note that you must check for JPEG\_REACHED\_SOS and

JPEG\_REACHED\_EOI return codes as the equivalent of jpeg\_read\_header's codes.

Once the first SOS marker has been reached, you must call

jpeg\_start\_decompress() before jpeg\_consume\_input() will consume more input;

it'll just keep returning JPEG\_REACHED\_SOS until you do. If you read a

tables-only file this way, jpeg\_consume\_input() will return JPEG\_REACHED\_EOI

without ever returning JPEG\_REACHED\_SOS; be sure to check for this case.

If this happens, the decompressor will not read any more input until you call

jpeg\_abort() to reset it. It is OK to call jpeg\_consume\_input() even when not

using buffered-image mode, but in that case it's basically a no-op after the

initial markers have been read: it will just return JPEG\_SUSPENDED.

Abbreviated datastreams and multiple images

-------------------------------------------

A JPEG compression or decompression object can be reused to process multiple

images. This saves a small amount of time per image by eliminating the

"create" and "destroy" operations, but that isn't the real purpose of the

feature. Rather, reuse of an object provides support for abbreviated JPEG

datastreams. Object reuse can also simplify processing a series of images in

a single input or output file. This section explains these features.

A JPEG file normally contains several hundred bytes worth of quantization

and Huffman tables. In a situation where many images will be stored or

transmitted with identical tables, this may represent an annoying overhead.

The JPEG standard therefore permits tables to be omitted. The standard

defines three classes of JPEG datastreams:

\* "Interchange" datastreams contain an image and all tables needed to decode

the image. These are the usual kind of JPEG file.

\* "Abbreviated image" datastreams contain an image, but are missing some or

all of the tables needed to decode that image.

\* "Abbreviated table specification" (henceforth "tables-only") datastreams

contain only table specifications.

To decode an abbreviated image, it is necessary to load the missing table(s)

into the decoder beforehand. This can be accomplished by reading a separate

tables-only file. A variant scheme uses a series of images in which the first

image is an interchange (complete) datastream, while subsequent ones are

abbreviated and rely on the tables loaded by the first image. It is assumed

that once the decoder has read a table, it will remember that table until a

new definition for the same table number is encountered.

It is the application designer's responsibility to figure out how to associate

the correct tables with an abbreviated image. While abbreviated datastreams

can be useful in a closed environment, their use is strongly discouraged in

any situation where data exchange with other applications might be needed.

Caveat designer.

The JPEG library provides support for reading and writing any combination of

tables-only datastreams and abbreviated images. In both compression and

decompression objects, a quantization or Huffman table will be retained for

the lifetime of the object, unless it is overwritten by a new table definition.

To create abbreviated image datastreams, it is only necessary to tell the

compressor not to emit some or all of the tables it is using. Each

quantization and Huffman table struct contains a boolean field "sent\_table",

which normally is initialized to FALSE. For each table used by the image, the

header-writing process emits the table and sets sent\_table = TRUE unless it is

already TRUE. (In normal usage, this prevents outputting the same table

definition multiple times, as would otherwise occur because the chroma

components typically share tables.) Thus, setting this field to TRUE before

calling jpeg\_start\_compress() will prevent the table from being written at

all.

If you want to create a "pure" abbreviated image file containing no tables,

just call "jpeg\_suppress\_tables(&cinfo, TRUE)" after constructing all the

tables. If you want to emit some but not all tables, you'll need to set the

individual sent\_table fields directly.

To create an abbreviated image, you must also call jpeg\_start\_compress()

with a second parameter of FALSE, not TRUE. Otherwise jpeg\_start\_compress()

will force all the sent\_table fields to FALSE. (This is a safety feature to

prevent abbreviated images from being created accidentally.)

To create a tables-only file, perform the same parameter setup that you

normally would, but instead of calling jpeg\_start\_compress() and so on, call

jpeg\_write\_tables(&cinfo). This will write an abbreviated datastream

containing only SOI, DQT and/or DHT markers, and EOI. All the quantization

and Huffman tables that are currently defined in the compression object will

be emitted unless their sent\_tables flag is already TRUE, and then all the

sent\_tables flags will be set TRUE.

A sure-fire way to create matching tables-only and abbreviated image files

is to proceed as follows:

create JPEG compression object

set JPEG parameters

set destination to tables-only file

jpeg\_write\_tables(&cinfo);

set destination to image file

jpeg\_start\_compress(&cinfo, FALSE);

write data...

jpeg\_finish\_compress(&cinfo);

Since the JPEG parameters are not altered between writing the table file and

the abbreviated image file, the same tables are sure to be used. Of course,

you can repeat the jpeg\_start\_compress() ... jpeg\_finish\_compress() sequence

many times to produce many abbreviated image files matching the table file.

You cannot suppress output of the computed Huffman tables when Huffman

optimization is selected. (If you could, there'd be no way to decode the

image...) Generally, you don't want to set optimize\_coding = TRUE when

you are trying to produce abbreviated files.

In some cases you might want to compress an image using tables which are

not stored in the application, but are defined in an interchange or

tables-only file readable by the application. This can be done by setting up

a JPEG decompression object to read the specification file, then copying the

tables into your compression object. See jpeg\_copy\_critical\_parameters()

for an example of copying quantization tables.

To read abbreviated image files, you simply need to load the proper tables

into the decompression object before trying to read the abbreviated image.

If the proper tables are stored in the application program, you can just

allocate the table structs and fill in their contents directly. For example,

to load a fixed quantization table into table slot "n":

if (cinfo.quant\_tbl\_ptrs[n] == NULL)

cinfo.quant\_tbl\_ptrs[n] = jpeg\_alloc\_quant\_table((j\_common\_ptr) &cinfo);

quant\_ptr = cinfo.quant\_tbl\_ptrs[n]; /\* quant\_ptr is JQUANT\_TBL\* \*/

for (i = 0; i < 64; i++) {

/\* Qtable[] is desired quantization table, in natural array order \*/

quant\_ptr->quantval[i] = Qtable[i];

}

Code to load a fixed Huffman table is typically (for AC table "n"):

if (cinfo.ac\_huff\_tbl\_ptrs[n] == NULL)

cinfo.ac\_huff\_tbl\_ptrs[n] = jpeg\_alloc\_huff\_table((j\_common\_ptr) &cinfo);

huff\_ptr = cinfo.ac\_huff\_tbl\_ptrs[n]; /\* huff\_ptr is JHUFF\_TBL\* \*/

for (i = 1; i <= 16; i++) {

/\* counts[i] is number of Huffman codes of length i bits, i=1..16 \*/

huff\_ptr->bits[i] = counts[i];

}

for (i = 0; i < 256; i++) {

/\* symbols[] is the list of Huffman symbols, in code-length order \*/

huff\_ptr->huffval[i] = symbols[i];

}

(Note that trying to set cinfo.quant\_tbl\_ptrs[n] to point directly at a

constant JQUANT\_TBL object is not safe. If the incoming file happened to

contain a quantization table definition, your master table would get

overwritten! Instead allocate a working table copy and copy the master table

into it, as illustrated above. Ditto for Huffman tables, of course.)

You might want to read the tables from a tables-only file, rather than

hard-wiring them into your application. The jpeg\_read\_header() call is

sufficient to read a tables-only file. You must pass a second parameter of

FALSE to indicate that you do not require an image to be present. Thus, the

typical scenario is

create JPEG decompression object

set source to tables-only file

jpeg\_read\_header(&cinfo, FALSE);

set source to abbreviated image file

jpeg\_read\_header(&cinfo, TRUE);

set decompression parameters

jpeg\_start\_decompress(&cinfo);

read data...

jpeg\_finish\_decompress(&cinfo);

In some cases, you may want to read a file without knowing whether it contains

an image or just tables. In that case, pass FALSE and check the return value

from jpeg\_read\_header(): it will be JPEG\_HEADER\_OK if an image was found,

JPEG\_HEADER\_TABLES\_ONLY if only tables were found. (A third return value,

JPEG\_SUSPENDED, is possible when using a suspending data source manager.)

Note that jpeg\_read\_header() will not complain if you read an abbreviated

image for which you haven't loaded the missing tables; the missing-table check

occurs later, in jpeg\_start\_decompress().

It is possible to read a series of images from a single source file by

repeating the jpeg\_read\_header() ... jpeg\_finish\_decompress() sequence,

without releasing/recreating the JPEG object or the data source module.

(If you did reinitialize, any partial bufferload left in the data source

buffer at the end of one image would be discarded, causing you to lose the

start of the next image.) When you use this method, stored tables are

automatically carried forward, so some of the images can be abbreviated images

that depend on tables from earlier images.

If you intend to write a series of images into a single destination file,

you might want to make a specialized data destination module that doesn't

flush the output buffer at term\_destination() time. This would speed things

up by some trifling amount. Of course, you'd need to remember to flush the

buffer after the last image. You can make the later images be abbreviated

ones by passing FALSE to jpeg\_start\_compress().

Special markers

---------------

Some applications may need to insert or extract special data in the JPEG

datastream. The JPEG standard provides marker types "COM" (comment) and

"APP0" through "APP15" (application) to hold application-specific data.

Unfortunately, the use of these markers is not specified by the standard.

COM markers are fairly widely used to hold user-supplied text. The JFIF file

format spec uses APP0 markers with specified initial strings to hold certain

data. Adobe applications use APP14 markers beginning with the string "Adobe"

for miscellaneous data. Other APPn markers are rarely seen, but might

contain almost anything.

If you wish to store user-supplied text, we recommend you use COM markers

and place readable 7-bit ASCII text in them. Newline conventions are not

standardized --- expect to find LF (Unix style), CR/LF (DOS style), or CR

(Mac style). A robust COM reader should be able to cope with random binary

garbage, including nulls, since some applications generate COM markers

containing non-ASCII junk. (But yours should not be one of them.)

For program-supplied data, use an APPn marker, and be sure to begin it with an

identifying string so that you can tell whether the marker is actually yours.

It's probably best to avoid using APP0 or APP14 for any private markers.

(NOTE: the upcoming SPIFF standard will use APP8 markers; we recommend you

not use APP8 markers for any private purposes, either.)

Keep in mind that at most 65533 bytes can be put into one marker, but you

can have as many markers as you like.

By default, the IJG compression library will write a JFIF APP0 marker if the

selected JPEG colorspace is grayscale or YCbCr, or an Adobe APP14 marker if

the selected colorspace is RGB, CMYK, or YCCK. You can disable this, but

we don't recommend it. The decompression library will recognize JFIF and

Adobe markers and will set the JPEG colorspace properly when one is found.

You can write special markers immediately following the datastream header by

calling jpeg\_write\_marker() after jpeg\_start\_compress() and before the first

call to jpeg\_write\_scanlines(). When you do this, the markers appear after

the SOI and the JFIF APP0 and Adobe APP14 markers (if written), but before

all else. Specify the marker type parameter as "JPEG\_COM" for COM or

"JPEG\_APP0 + n" for APPn. (Actually, jpeg\_write\_marker will let you write

any marker type, but we don't recommend writing any other kinds of marker.)

For example, to write a user comment string pointed to by comment\_text:

jpeg\_write\_marker(cinfo, JPEG\_COM, comment\_text, strlen(comment\_text));

If it's not convenient to store all the marker data in memory at once,

you can instead call jpeg\_write\_m\_header() followed by multiple calls to

jpeg\_write\_m\_byte(). If you do it this way, it's your responsibility to

call jpeg\_write\_m\_byte() exactly the number of times given in the length

parameter to jpeg\_write\_m\_header(). (This method lets you empty the

output buffer partway through a marker, which might be important when

using a suspending data destination module. In any case, if you are using

a suspending destination, you should flush its buffer after inserting

any special markers. See "I/O suspension".)

Or, if you prefer to synthesize the marker byte sequence yourself,

you can just cram it straight into the data destination module.

If you are writing JFIF 1.02 extension markers (thumbnail images), don't

forget to set cinfo.JFIF\_minor\_version = 2 so that the encoder will write the

correct JFIF version number in the JFIF header marker. The library's default

is to write version 1.01, but that's wrong if you insert any 1.02 extension

markers. (We could probably get away with just defaulting to 1.02, but there

used to be broken decoders that would complain about unknown minor version

numbers. To reduce compatibility risks it's safest not to write 1.02 unless

you are actually using 1.02 extensions.)

When reading, two methods of handling special markers are available:

1. You can ask the library to save the contents of COM and/or APPn markers

into memory, and then examine them at your leisure afterwards.

2. You can supply your own routine to process COM and/or APPn markers

on-the-fly as they are read.

The first method is simpler to use, especially if you are using a suspending

data source; writing a marker processor that copes with input suspension is

not easy (consider what happens if the marker is longer than your available

input buffer). However, the second method conserves memory since the marker

data need not be kept around after it's been processed.

For either method, you'd normally set up marker handling after creating a

decompression object and before calling jpeg\_read\_header(), because the

markers of interest will typically be near the head of the file and so will

be scanned by jpeg\_read\_header. Once you've established a marker handling

method, it will be used for the life of that decompression object

(potentially many datastreams), unless you change it. Marker handling is

determined separately for COM markers and for each APPn marker code.

To save the contents of special markers in memory, call

jpeg\_save\_markers(cinfo, marker\_code, length\_limit)

where marker\_code is the marker type to save, JPEG\_COM or JPEG\_APP0+n.

(To arrange to save all the special marker types, you need to call this

routine 17 times, for COM and APP0-APP15.) If the incoming marker is longer

than length\_limit data bytes, only length\_limit bytes will be saved; this

parameter allows you to avoid chewing up memory when you only need to see the

first few bytes of a potentially large marker. If you want to save all the

data, set length\_limit to 0xFFFF; that is enough since marker lengths are only

16 bits. As a special case, setting length\_limit to 0 prevents that marker

type from being saved at all. (That is the default behavior, in fact.)

After jpeg\_read\_header() completes, you can examine the special markers by

following the cinfo->marker\_list pointer chain. All the special markers in

the file appear in this list, in order of their occurrence in the file (but

omitting any markers of types you didn't ask for). Both the original data

length and the saved data length are recorded for each list entry; the latter

will not exceed length\_limit for the particular marker type. Note that these

lengths exclude the marker length word, whereas the stored representation

within the JPEG file includes it. (Hence the maximum data length is really

only 65533.)

It is possible that additional special markers appear in the file beyond the

SOS marker at which jpeg\_read\_header stops; if so, the marker list will be

extended during reading of the rest of the file. This is not expected to be

common, however. If you are short on memory you may want to reset the length

limit to zero for all marker types after finishing jpeg\_read\_header, to

ensure that the max\_memory\_to\_use setting cannot be exceeded due to addition

of later markers.

The marker list remains stored until you call jpeg\_finish\_decompress or

jpeg\_abort, at which point the memory is freed and the list is set to empty.

(jpeg\_destroy also releases the storage, of course.)

Note that the library is internally interested in APP0 and APP14 markers;

if you try to set a small nonzero length limit on these types, the library

will silently force the length up to the minimum it wants. (But you can set

a zero length limit to prevent them from being saved at all.) Also, in a

16-bit environment, the maximum length limit may be constrained to less than

65533 by malloc() limitations. It is therefore best not to assume that the

effective length limit is exactly what you set it to be.

If you want to supply your own marker-reading routine, you do it by calling

jpeg\_set\_marker\_processor(). A marker processor routine must have the

signature

boolean jpeg\_marker\_parser\_method (j\_decompress\_ptr cinfo)

Although the marker code is not explicitly passed, the routine can find it

in cinfo->unread\_marker. At the time of call, the marker proper has been

read from the data source module. The processor routine is responsible for

reading the marker length word and the remaining parameter bytes, if any.

Return TRUE to indicate success. (FALSE should be returned only if you are

using a suspending data source and it tells you to suspend. See the standard

marker processors in jdmarker.c for appropriate coding methods if you need to

use a suspending data source.)

If you override the default APP0 or APP14 processors, it is up to you to

recognize JFIF and Adobe markers if you want colorspace recognition to occur

properly. We recommend copying and extending the default processors if you

want to do that. (A better idea is to save these marker types for later

examination by calling jpeg\_save\_markers(); that method doesn't interfere

with the library's own processing of these markers.)

jpeg\_set\_marker\_processor() and jpeg\_save\_markers() are mutually exclusive

--- if you call one it overrides any previous call to the other, for the

particular marker type specified.

A simple example of an external COM processor can be found in djpeg.c.

Also, see jpegtran.c for an example of using jpeg\_save\_markers.

Raw (downsampled) image data

----------------------------

Some applications need to supply already-downsampled image data to the JPEG

compressor, or to receive raw downsampled data from the decompressor. The

library supports this requirement by allowing the application to write or

read raw data, bypassing the normal preprocessing or postprocessing steps.

The interface is different from the standard one and is somewhat harder to

use. If your interest is merely in bypassing color conversion, we recommend

that you use the standard interface and simply set jpeg\_color\_space =

in\_color\_space (or jpeg\_color\_space = out\_color\_space for decompression).

The mechanism described in this section is necessary only to supply or

receive downsampled image data, in which not all components have the same

dimensions.

To compress raw data, you must supply the data in the colorspace to be used

in the JPEG file (please read the earlier section on Special color spaces)

and downsampled to the sampling factors specified in the JPEG parameters.

You must supply the data in the format used internally by the JPEG library,

namely a JSAMPIMAGE array. This is an array of pointers to two-dimensional

arrays, each of type JSAMPARRAY. Each 2-D array holds the values for one

color component. This structure is necessary since the components are of

different sizes. If the image dimensions are not a multiple of the MCU size,

you must also pad the data correctly (usually, this is done by replicating

the last column and/or row). The data must be padded to a multiple of a DCT

block in each component: that is, each downsampled row must contain a

multiple of 8 valid samples, and there must be a multiple of 8 sample rows

for each component. (For applications such as conversion of digital TV

images, the standard image size is usually a multiple of the DCT block size,

so that no padding need actually be done.)

The procedure for compression of raw data is basically the same as normal

compression, except that you call jpeg\_write\_raw\_data() in place of

jpeg\_write\_scanlines(). Before calling jpeg\_start\_compress(), you must do

the following:

\* Set cinfo->raw\_data\_in to TRUE. (It is set FALSE by jpeg\_set\_defaults().)

This notifies the library that you will be supplying raw data.

\* Ensure jpeg\_color\_space is correct --- an explicit jpeg\_set\_colorspace()

call is a good idea. Note that since color conversion is bypassed,

in\_color\_space is ignored, except that jpeg\_set\_defaults() uses it to

choose the default jpeg\_color\_space setting.

\* Ensure the sampling factors, cinfo->comp\_info[i].h\_samp\_factor and

cinfo->comp\_info[i].v\_samp\_factor, are correct. Since these indicate the

dimensions of the data you are supplying, it's wise to set them

explicitly, rather than assuming the library's defaults are what you want.

To pass raw data to the library, call jpeg\_write\_raw\_data() in place of

jpeg\_write\_scanlines(). The two routines work similarly except that

jpeg\_write\_raw\_data takes a JSAMPIMAGE data array rather than JSAMPARRAY.

The scanlines count passed to and returned from jpeg\_write\_raw\_data is

measured in terms of the component with the largest v\_samp\_factor.

jpeg\_write\_raw\_data() processes one MCU row per call, which is to say

v\_samp\_factor\*DCTSIZE sample rows of each component. The passed num\_lines

value must be at least max\_v\_samp\_factor\*DCTSIZE, and the return value will

be exactly that amount (or possibly some multiple of that amount, in future

library versions). This is true even on the last call at the bottom of the

image; don't forget to pad your data as necessary.

The required dimensions of the supplied data can be computed for each

component as

cinfo->comp\_info[i].width\_in\_blocks\*DCTSIZE samples per row

cinfo->comp\_info[i].height\_in\_blocks\*DCTSIZE rows in image

after jpeg\_start\_compress() has initialized those fields. If the valid data

is smaller than this, it must be padded appropriately. For some sampling

factors and image sizes, additional dummy DCT blocks are inserted to make

the image a multiple of the MCU dimensions. The library creates such dummy

blocks itself; it does not read them from your supplied data. Therefore you

need never pad by more than DCTSIZE samples. An example may help here.

Assume 2h2v downsampling of YCbCr data, that is

cinfo->comp\_info[0].h\_samp\_factor = 2 for Y

cinfo->comp\_info[0].v\_samp\_factor = 2

cinfo->comp\_info[1].h\_samp\_factor = 1 for Cb

cinfo->comp\_info[1].v\_samp\_factor = 1

cinfo->comp\_info[2].h\_samp\_factor = 1 for Cr

cinfo->comp\_info[2].v\_samp\_factor = 1

and suppose that the nominal image dimensions (cinfo->image\_width and

cinfo->image\_height) are 101x101 pixels. Then jpeg\_start\_compress() will

compute downsampled\_width = 101 and width\_in\_blocks = 13 for Y,

downsampled\_width = 51 and width\_in\_blocks = 7 for Cb and Cr (and the same

for the height fields). You must pad the Y data to at least 13\*8 = 104

columns and rows, the Cb/Cr data to at least 7\*8 = 56 columns and rows. The

MCU height is max\_v\_samp\_factor = 2 DCT rows so you must pass at least 16

scanlines on each call to jpeg\_write\_raw\_data(), which is to say 16 actual

sample rows of Y and 8 each of Cb and Cr. A total of 7 MCU rows are needed,

so you must pass a total of 7\*16 = 112 "scanlines". The last DCT block row

of Y data is dummy, so it doesn't matter what you pass for it in the data

arrays, but the scanlines count must total up to 112 so that all of the Cb

and Cr data gets passed.

Output suspension is supported with raw-data compression: if the data

destination module suspends, jpeg\_write\_raw\_data() will return 0.

In this case the same data rows must be passed again on the next call.

Decompression with raw data output implies bypassing all postprocessing:

you cannot ask for rescaling or color quantization, for instance. More

seriously, you must deal with the color space and sampling factors present in

the incoming file. If your application only handles, say, 2h1v YCbCr data,

you must check for and fail on other color spaces or other sampling factors.

The library will not convert to a different color space for you.

To obtain raw data output, set cinfo->raw\_data\_out = TRUE before

jpeg\_start\_decompress() (it is set FALSE by jpeg\_read\_header()). Be sure to

verify that the color space and sampling factors are ones you can handle.

Then call jpeg\_read\_raw\_data() in place of jpeg\_read\_scanlines(). The

decompression process is otherwise the same as usual.

jpeg\_read\_raw\_data() returns one MCU row per call, and thus you must pass a

buffer of at least max\_v\_samp\_factor\*DCTSIZE scanlines (scanline counting is

the same as for raw-data compression). The buffer you pass must be large

enough to hold the actual data plus padding to DCT-block boundaries. As with

compression, any entirely dummy DCT blocks are not processed so you need not

allocate space for them, but the total scanline count includes them. The

above example of computing buffer dimensions for raw-data compression is

equally valid for decompression.

Input suspension is supported with raw-data decompression: if the data source

module suspends, jpeg\_read\_raw\_data() will return 0. You can also use

buffered-image mode to read raw data in multiple passes.

Really raw data: DCT coefficients

---------------------------------

It is possible to read or write the contents of a JPEG file as raw DCT

coefficients. This facility is mainly intended for use in lossless

transcoding between different JPEG file formats. Other possible applications

include lossless cropping of a JPEG image, lossless reassembly of a

multi-strip or multi-tile TIFF/JPEG file into a single JPEG datastream, etc.

To read the contents of a JPEG file as DCT coefficients, open the file and do

jpeg\_read\_header() as usual. But instead of calling jpeg\_start\_decompress()

and jpeg\_read\_scanlines(), call jpeg\_read\_coefficients(). This will read the

entire image into a set of virtual coefficient-block arrays, one array per

component. The return value is a pointer to an array of virtual-array

descriptors. Each virtual array can be accessed directly using the JPEG

memory manager's access\_virt\_barray method (see Memory management, below,

and also read structure.doc's discussion of virtual array handling). Or,

for simple transcoding to a different JPEG file format, the array list can

just be handed directly to jpeg\_write\_coefficients().

Each block in the block arrays contains quantized coefficient values in

normal array order (not JPEG zigzag order). The block arrays contain only

DCT blocks containing real data; any entirely-dummy blocks added to fill out

interleaved MCUs at the right or bottom edges of the image are discarded

during reading and are not stored in the block arrays. (The size of each

block array can be determined from the width\_in\_blocks and height\_in\_blocks

fields of the component's comp\_info entry.) This is also the data format

expected by jpeg\_write\_coefficients().

When you are done using the virtual arrays, call jpeg\_finish\_decompress()

to release the array storage and return the decompression object to an idle

state; or just call jpeg\_destroy() if you don't need to reuse the object.

If you use a suspending data source, jpeg\_read\_coefficients() will return

NULL if it is forced to suspend; a non-NULL return value indicates successful

completion. You need not test for a NULL return value when using a

non-suspending data source.

It is also possible to call jpeg\_read\_coefficients() to obtain access to the

decoder's coefficient arrays during a normal decode cycle in buffered-image

mode. This frammish might be useful for progressively displaying an incoming

image and then re-encoding it without loss. To do this, decode in buffered-

image mode as discussed previously, then call jpeg\_read\_coefficients() after

the last jpeg\_finish\_output() call. The arrays will be available for your use

until you call jpeg\_finish\_decompress().

To write the contents of a JPEG file as DCT coefficients, you must provide

the DCT coefficients stored in virtual block arrays. You can either pass

block arrays read from an input JPEG file by jpeg\_read\_coefficients(), or

allocate virtual arrays from the JPEG compression object and fill them

yourself. In either case, jpeg\_write\_coefficients() is substituted for

jpeg\_start\_compress() and jpeg\_write\_scanlines(). Thus the sequence is

\* Create compression object

\* Set all compression parameters as necessary

\* Request virtual arrays if needed

\* jpeg\_write\_coefficients()

\* jpeg\_finish\_compress()

\* Destroy or re-use compression object

jpeg\_write\_coefficients() is passed a pointer to an array of virtual block

array descriptors; the number of arrays is equal to cinfo.num\_components.

The virtual arrays need only have been requested, not realized, before

jpeg\_write\_coefficients() is called. A side-effect of

jpeg\_write\_coefficients() is to realize any virtual arrays that have been

requested from the compression object's memory manager. Thus, when obtaining

the virtual arrays from the compression object, you should fill the arrays

after calling jpeg\_write\_coefficients(). The data is actually written out

when you call jpeg\_finish\_compress(); jpeg\_write\_coefficients() only writes

the file header.

When writing raw DCT coefficients, it is crucial that the JPEG quantization

tables and sampling factors match the way the data was encoded, or the

resulting file will be invalid. For transcoding from an existing JPEG file,

we recommend using jpeg\_copy\_critical\_parameters(). This routine initializes

all the compression parameters to default values (like jpeg\_set\_defaults()),

then copies the critical information from a source decompression object.

The decompression object should have just been used to read the entire

JPEG input file --- that is, it should be awaiting jpeg\_finish\_decompress().

jpeg\_write\_coefficients() marks all tables stored in the compression object

as needing to be written to the output file (thus, it acts like

jpeg\_start\_compress(cinfo, TRUE)). This is for safety's sake, to avoid

emitting abbreviated JPEG files by accident. If you really want to emit an

abbreviated JPEG file, call jpeg\_suppress\_tables(), or set the tables'

individual sent\_table flags, between calling jpeg\_write\_coefficients() and

jpeg\_finish\_compress().

Progress monitoring

-------------------

Some applications may need to regain control from the JPEG library every so

often. The typical use of this feature is to produce a percent-done bar or

other progress display. (For a simple example, see cjpeg.c or djpeg.c.)

Although you do get control back frequently during the data-transferring pass

(the jpeg\_read\_scanlines or jpeg\_write\_scanlines loop), any additional passes

will occur inside jpeg\_finish\_compress or jpeg\_start\_decompress; those

routines may take a long time to execute, and you don't get control back

until they are done.

You can define a progress-monitor routine which will be called periodically

by the library. No guarantees are made about how often this call will occur,

so we don't recommend you use it for mouse tracking or anything like that.

At present, a call will occur once per MCU row, scanline, or sample row

group, whichever unit is convenient for the current processing mode; so the

wider the image, the longer the time between calls. During the data

transferring pass, only one call occurs per call of jpeg\_read\_scanlines or

jpeg\_write\_scanlines, so don't pass a large number of scanlines at once if

you want fine resolution in the progress count. (If you really need to use

the callback mechanism for time-critical tasks like mouse tracking, you could

insert additional calls inside some of the library's inner loops.)

To establish a progress-monitor callback, create a struct jpeg\_progress\_mgr,

fill in its progress\_monitor field with a pointer to your callback routine,

and set cinfo->progress to point to the struct. The callback will be called

whenever cinfo->progress is non-NULL. (This pointer is set to NULL by

jpeg\_create\_compress or jpeg\_create\_decompress; the library will not change

it thereafter. So if you allocate dynamic storage for the progress struct,

make sure it will live as long as the JPEG object does. Allocating from the

JPEG memory manager with lifetime JPOOL\_PERMANENT will work nicely.) You

can use the same callback routine for both compression and decompression.

The jpeg\_progress\_mgr struct contains four fields which are set by the library:

long pass\_counter; /\* work units completed in this pass \*/

long pass\_limit; /\* total number of work units in this pass \*/

int completed\_passes; /\* passes completed so far \*/

int total\_passes; /\* total number of passes expected \*/

During any one pass, pass\_counter increases from 0 up to (not including)

pass\_limit; the step size is usually but not necessarily 1. The pass\_limit

value may change from one pass to another. The expected total number of

passes is in total\_passes, and the number of passes already completed is in

completed\_passes. Thus the fraction of work completed may be estimated as

completed\_passes + (pass\_counter/pass\_limit)

--------------------------------------------

total\_passes

ignoring the fact that the passes may not be equal amounts of work.

When decompressing, pass\_limit can even change within a pass, because it

depends on the number of scans in the JPEG file, which isn't always known in

advance. The computed fraction-of-work-done may jump suddenly (if the library

discovers it has overestimated the number of scans) or even decrease (in the

opposite case). It is not wise to put great faith in the work estimate.

When using the decompressor's buffered-image mode, the progress monitor work

estimate is likely to be completely unhelpful, because the library has no way

to know how many output passes will be demanded of it. Currently, the library

sets total\_passes based on the assumption that there will be one more output

pass if the input file end hasn't yet been read (jpeg\_input\_complete() isn't

TRUE), but no more output passes if the file end has been reached when the

output pass is started. This means that total\_passes will rise as additional

output passes are requested. If you have a way of determining the input file

size, estimating progress based on the fraction of the file that's been read

will probably be more useful than using the library's value.

Memory management

-----------------

This section covers some key facts about the JPEG library's built-in memory

manager. For more info, please read structure.doc's section about the memory

manager, and consult the source code if necessary.

All memory and temporary file allocation within the library is done via the

memory manager. If necessary, you can replace the "back end" of the memory

manager to control allocation yourself (for example, if you don't want the

library to use malloc() and free() for some reason).

Some data is allocated "permanently" and will not be freed until the JPEG

object is destroyed. Most data is allocated "per image" and is freed by

jpeg\_finish\_compress, jpeg\_finish\_decompress, or jpeg\_abort. You can call the

memory manager yourself to allocate structures that will automatically be

freed at these times. Typical code for this is

ptr = (\*cinfo->mem->alloc\_small) ((j\_common\_ptr) cinfo, JPOOL\_IMAGE, size);

Use JPOOL\_PERMANENT to get storage that lasts as long as the JPEG object.

Use alloc\_large instead of alloc\_small for anything bigger than a few Kbytes.

There are also alloc\_sarray and alloc\_barray routines that automatically

build 2-D sample or block arrays.

The library's minimum space requirements to process an image depend on the

image's width, but not on its height, because the library ordinarily works

with "strip" buffers that are as wide as the image but just a few rows high.

Some operating modes (eg, two-pass color quantization) require full-image

buffers. Such buffers are treated as "virtual arrays": only the current strip

need be in memory, and the rest can be swapped out to a temporary file.

If you use the simplest memory manager back end (jmemnobs.c), then no

temporary files are used; virtual arrays are simply malloc()'d. Images bigger

than memory can be processed only if your system supports virtual memory.

The other memory manager back ends support temporary files of various flavors

and thus work in machines without virtual memory. They may also be useful on

Unix machines if you need to process images that exceed available swap space.

When using temporary files, the library will make the in-memory buffers for

its virtual arrays just big enough to stay within a "maximum memory" setting.

Your application can set this limit by setting cinfo->mem->max\_memory\_to\_use

after creating the JPEG object. (Of course, there is still a minimum size for

the buffers, so the max-memory setting is effective only if it is bigger than

the minimum space needed.) If you allocate any large structures yourself, you

must allocate them before jpeg\_start\_compress() or jpeg\_start\_decompress() in

order to have them counted against the max memory limit. Also keep in mind

that space allocated with alloc\_small() is ignored, on the assumption that

it's too small to be worth worrying about; so a reasonable safety margin

should be left when setting max\_memory\_to\_use.

If you use the jmemname.c or jmemdos.c memory manager back end, it is

important to clean up the JPEG object properly to ensure that the temporary

files get deleted. (This is especially crucial with jmemdos.c, where the

"temporary files" may be extended-memory segments; if they are not freed,

DOS will require a reboot to recover the memory.) Thus, with these memory

managers, it's a good idea to provide a signal handler that will trap any

early exit from your program. The handler should call either jpeg\_abort()

or jpeg\_destroy() for any active JPEG objects. A handler is not needed with

jmemnobs.c, and shouldn't be necessary with jmemansi.c or jmemmac.c either,

since the C library is supposed to take care of deleting files made with

tmpfile().

Memory usage

------------

Working memory requirements while performing compression or decompression

depend on image dimensions, image characteristics (such as colorspace and

JPEG process), and operating mode (application-selected options).

As of v6b, the decompressor requires:

1. About 24K in more-or-less-fixed-size data. This varies a bit depending

on operating mode and image characteristics (particularly color vs.

grayscale), but it doesn't depend on image dimensions.

2. Strip buffers (of size proportional to the image width) for IDCT and

upsampling results. The worst case for commonly used sampling factors

is about 34 bytes \* width in pixels for a color image. A grayscale image

only needs about 8 bytes per pixel column.

3. A full-image DCT coefficient buffer is needed to decode a multi-scan JPEG

file (including progressive JPEGs), or whenever you select buffered-image

mode. This takes 2 bytes/coefficient. At typical 2x2 sampling, that's

3 bytes per pixel for a color image. Worst case (1x1 sampling) requires

6 bytes/pixel. For grayscale, figure 2 bytes/pixel.

4. To perform 2-pass color quantization, the decompressor also needs a

128K color lookup table and a full-image pixel buffer (3 bytes/pixel).

This does not count any memory allocated by the application, such as a

buffer to hold the final output image.

The above figures are valid for 8-bit JPEG data precision and a machine with

32-bit ints. For 12-bit JPEG data, double the size of the strip buffers and

quantization pixel buffer. The "fixed-size" data will be somewhat smaller

with 16-bit ints, larger with 64-bit ints. Also, CMYK or other unusual

color spaces will require different amounts of space.

The full-image coefficient and pixel buffers, if needed at all, do not

have to be fully RAM resident; you can have the library use temporary

files instead when the total memory usage would exceed a limit you set.

(But if your OS supports virtual memory, it's probably better to just use

jmemnobs and let the OS do the swapping.)

The compressor's memory requirements are similar, except that it has no need

for color quantization. Also, it needs a full-image DCT coefficient buffer

if Huffman-table optimization is asked for, even if progressive mode is not

requested.

If you need more detailed information about memory usage in a particular

situation, you can enable the MEM\_STATS code in jmemmgr.c.

Library compile-time options

----------------------------

A number of compile-time options are available by modifying jmorecfg.h.

The JPEG standard provides for both the baseline 8-bit DCT process and

a 12-bit DCT process. The IJG code supports 12-bit lossy JPEG if you define

BITS\_IN\_JSAMPLE as 12 rather than 8. Note that this causes JSAMPLE to be

larger than a char, so it affects the surrounding application's image data.

The sample applications cjpeg and djpeg can support 12-bit mode only for PPM

and GIF file formats; you must disable the other file formats to compile a

12-bit cjpeg or djpeg. (install.doc has more information about that.)

At present, a 12-bit library can handle \*only\* 12-bit images, not both

precisions. (If you need to include both 8- and 12-bit libraries in a single

application, you could probably do it by defining NEED\_SHORT\_EXTERNAL\_NAMES

for just one of the copies. You'd have to access the 8-bit and 12-bit copies

from separate application source files. This is untested ... if you try it,

we'd like to hear whether it works!)

Note that a 12-bit library always compresses in Huffman optimization mode,

in order to generate valid Huffman tables. This is necessary because our

default Huffman tables only cover 8-bit data. If you need to output 12-bit

files in one pass, you'll have to supply suitable default Huffman tables.

You may also want to supply your own DCT quantization tables; the existing

quality-scaling code has been developed for 8-bit use, and probably doesn't

generate especially good tables for 12-bit.

The maximum number of components (color channels) in the image is determined

by MAX\_COMPONENTS. The JPEG standard allows up to 255 components, but we

expect that few applications will need more than four or so.

On machines with unusual data type sizes, you may be able to improve

performance or reduce memory space by tweaking the various typedefs in

jmorecfg.h. In particular, on some RISC CPUs, access to arrays of "short"s

is quite slow; consider trading memory for speed by making JCOEF, INT16, and

UINT16 be "int" or "unsigned int". UINT8 is also a candidate to become int.

You probably don't want to make JSAMPLE be int unless you have lots of memory

to burn.

You can reduce the size of the library by compiling out various optional

functions. To do this, undefine xxx\_SUPPORTED symbols as necessary.

You can also save a few K by not having text error messages in the library;

the standard error message table occupies about 5Kb. This is particularly

reasonable for embedded applications where there's no good way to display

a message anyway. To do this, remove the creation of the message table

(jpeg\_std\_message\_table[]) from jerror.c, and alter format\_message to do

something reasonable without it. You could output the numeric value of the

message code number, for example. If you do this, you can also save a couple

more K by modifying the TRACEMSn() macros in jerror.h to expand to nothing;

you don't need trace capability anyway, right?

Portability considerations

--------------------------

The JPEG library has been written to be extremely portable; the sample

applications cjpeg and djpeg are slightly less so. This section summarizes

the design goals in this area. (If you encounter any bugs that cause the

library to be less portable than is claimed here, we'd appreciate hearing

about them.)

The code works fine on ANSI C, C++, and pre-ANSI C compilers, using any of

the popular system include file setups, and some not-so-popular ones too.

See install.doc for configuration procedures.

The code is not dependent on the exact sizes of the C data types. As

distributed, we make the assumptions that

char is at least 8 bits wide

short is at least 16 bits wide

int is at least 16 bits wide

long is at least 32 bits wide

(These are the minimum requirements of the ANSI C standard.) Wider types will

work fine, although memory may be used inefficiently if char is much larger

than 8 bits or short is much bigger than 16 bits. The code should work

equally well with 16- or 32-bit ints.

In a system where these assumptions are not met, you may be able to make the

code work by modifying the typedefs in jmorecfg.h. However, you will probably

have difficulty if int is less than 16 bits wide, since references to plain

int abound in the code.

char can be either signed or unsigned, although the code runs faster if an

unsigned char type is available. If char is wider than 8 bits, you will need

to redefine JOCTET and/or provide custom data source/destination managers so

that JOCTET represents exactly 8 bits of data on external storage.

The JPEG library proper does not assume ASCII representation of characters.

But some of the image file I/O modules in cjpeg/djpeg do have ASCII

dependencies in file-header manipulation; so does cjpeg's select\_file\_type()

routine.

The JPEG library does not rely heavily on the C library. In particular, C

stdio is used only by the data source/destination modules and the error

handler, all of which are application-replaceable. (cjpeg/djpeg are more

heavily dependent on stdio.) malloc and free are called only from the memory

manager "back end" module, so you can use a different memory allocator by

replacing that one file.

The code generally assumes that C names must be unique in the first 15

characters. However, global function names can be made unique in the

first 6 characters by defining NEED\_SHORT\_EXTERNAL\_NAMES.

More info about porting the code may be gleaned by reading jconfig.doc,

jmorecfg.h, and jinclude.h.

Notes for MS-DOS implementors

-----------------------------

The IJG code is designed to work efficiently in 80x86 "small" or "medium"

memory models (i.e., data pointers are 16 bits unless explicitly declared

"far"; code pointers can be either size). You may be able to use small

model to compile cjpeg or djpeg by itself, but you will probably have to use

medium model for any larger application. This won't make much difference in

performance. You \*will\* take a noticeable performance hit if you use a

large-data memory model (perhaps 10%-25%), and you should avoid "huge" model

if at all possible.

The JPEG library typically needs 2Kb-3Kb of stack space. It will also

malloc about 20K-30K of near heap space while executing (and lots of far

heap, but that doesn't count in this calculation). This figure will vary

depending on selected operating mode, and to a lesser extent on image size.

There is also about 5Kb-6Kb of constant data which will be allocated in the

near data segment (about 4Kb of this is the error message table).

Thus you have perhaps 20K available for other modules' static data and near

heap space before you need to go to a larger memory model. The C library's

static data will account for several K of this, but that still leaves a good

deal for your needs. (If you are tight on space, you could reduce the sizes

of the I/O buffers allocated by jdatasrc.c and jdatadst.c, say from 4K to

1K. Another possibility is to move the error message table to far memory;

this should be doable with only localized hacking on jerror.c.)

About 2K of the near heap space is "permanent" memory that will not be

released until you destroy the JPEG object. This is only an issue if you

save a JPEG object between compression or decompression operations.

Far data space may also be a tight resource when you are dealing with large

images. The most memory-intensive case is decompression with two-pass color

quantization, or single-pass quantization to an externally supplied color

map. This requires a 128Kb color lookup table plus strip buffers amounting

to about 40 bytes per column for typical sampling ratios (eg, about 25600

bytes for a 640-pixel-wide image). You may not be able to process wide

images if you have large data structures of your own.

Of course, all of these concerns vanish if you use a 32-bit flat-memory-model

compiler, such as DJGPP or Watcom C. We highly recommend flat model if you

can use it; the JPEG library is significantly faster in flat model.