

Intel[®] JPEG Library

Porting to the IJL from the
Independent JPEG Group's JPEG
Source



Version 1.3
August 15, 1998

Copyright © 1998 Intel Corporation. All rights reserved.
Intel Corporation, 5200 N.E. Elam Young Parkway, Hillsboro, OR 97124-6497

Intel Corporation assumes no responsibility for errors or omissions in this guide. Nor does Intel make any commitment to update the information contained herein.

* Other product and corporate names may be trademarks of other companies and are used only for explanation and the owners' benefit, without intent to infringe.

Table of Contents

1. Introduction	1
1.1 About This Document	1
1.2 Nature of Product	1
1.3 Technical Support and Feedback	1
2. Overview	2
2.1 Matching the IJG Sample Application “djpeg.exe”	3
2.2 Matching the IJG Sample Application “cjpeg.exe”	13
2.3 Quality differences between IJL and IJG encoder quality settings.....	21
3. Appendix A: A Memory-Mapped Bitmap implementation of a JPEG Decoder	22
4. Appendix B: A Memory-Mapped Bitmap implementation of a JPEG Encoder	25

This page is intentionally left blank. Needed for two-sided printing.

This page is intentionally left blank. Needed for two-sided printing.

1. Introduction

1.1 About This Document

This document provides help to developers who wish to port an application currently interfacing with the Independent JPEG Group's (IJG) version 6a JPEG codec to one using the high performance Intel® JPEG Library (IJL).

It is assumed that the reader has a working knowledge of the software development process and the C/C++ programming language. The reader should be familiar with the IJG version 6a JPEG codec. Some familiarity with digital imaging, software development for the Microsoft* Windows* 95 operating system, and the Microsoft Foundation Classes* application framework may also be useful.

1.2 Nature of Product

The IJL is a software library for application developers that provides high performance JPEG encoding and decoding of full color, and grayscale, continuous-tone still images.

The IJL was designed for use on Intel® architecture platforms and has been tuned for high performance and efficient memory usage. Additionally, the IJL was developed to take advantage of MMX™ technology if present.

The IJL provides an easy-to-use programming interface without sacrificing low-level JPEG control to advanced developers. The IJL also includes a substantial amount of functionality that is not included in the ISO JPEG standard. This added functionality is typically necessary when working with JPEG images, and includes pre-processing and post-processing options like sampling and color space conversions.

1.3 Technical Support and Feedback

Your feedback on the IJL is very important to us. We will strive to provide you with answers or solutions to any problems you might encounter. To give your feedback, or to report any problems with installation or use, please contact one of the following:

- Support Hotlines:
North American Hotline: 800-628-8686
International Hotline: 916-356-7599
- Send e-mail to developer_support@intel.com

2. Overview

The Intel® JPEG Library (IJL) is a robust, high performance JPEG engine. It was designed with a simple application interface and well-defined usage models. In contrast, the Independent JPEG Group’s (IJG) version 6a JPEG codec is a code-level, platform independent, reference JPEG implementation. It was one of the original implementations of the JPEG standard and was influential in JPEG’s adoption. The IJG implementation has withstood the test of time because it is freely available, and it has benefited from years of use and feedback.

The IJG implementation does not have a well-defined (i.e., strict) user interface. It was written in the “C” programming language. It needs to be included in a program, or “packaged” by a programmer, prior to its use. It is not a library, or a binary object, or any other executable primitive like the IJL. However, several sample applications included with the public version of IJG version 6a have implemented common JPEG usage models and illustrate the use of the IJG implementation in common scenarios.

It is from these examples that we may draw comparisons to the IJL. The interface to the IJL and IJG is fundamentally different (the IJL has a well-defined API whereas the IJG has only a C source framework). While no one can anticipate all possible uses of a JPEG implementation, the IJL is capable of most of the functionality provided by the IJG code base. The following table illustrates the similarities and differences between the two.

Feature	IJL Supported?	IJG Supported?
Decode an image by scanlines	Yes	Yes
Encode an entire JPEG image	Yes	Yes
Support bottom-up bitmaps	Yes	Yes
Support Custom JPEG tables (i.e., Huffman and quantization)	Yes	Yes
Support variable quality JPEG authoring	Yes	Yes
Decode an arbitrary region of an image	Yes	No
Support custom bitmap formats (i.e., BGR and RGB)	Yes	No
Support interrupted processing	Yes	No
Encode an image by scanlines	No	Yes
Support custom error handling	No	Yes
Support user-customized JPEG processing elements	No	Yes

Feature	IJL Supported?	IJG Supported?
Support little-endian byte order architectures	No	Yes
Support Progressive JPEG encoding	No	Yes
Support decoding of GIF, PPM, BMP, TGA, RLE file formats	No	Yes
Support for palettized output	No	Yes
Optimal Huffman Table generation	No	Yes
Support for JPEG recompression	No	Yes
Support for CMYK JPEG Images	No	Yes
Support 12-bit JPEG decoding	No	Yes [†]

Table 1 Comparing & Contrasting the IJL and IJG.

[†]Supports either 8 bit or 12 bit, not both.

2.1 Matching the IJG Sample Application “djpeg.exe”

An important application included with the IJG source code is a DOS* executable that permits command-line control over many of the decoding features presented by IJG. We will replace the core components of this code by the IJL equivalents.

The following code is an adapted version of the “djpeg” executable included with IJG 6a. This application uses IJG to decode a JPEG image to a Windows* DIB file.

```
//Note: This code is based on a file "djpeg.c" from the Independent
//JPEG Group (IJG) JPEG codec Version 6a. Please see the original
//IJG README file for legal information relating to code distribution
//and attribution

//IJG Copyright:
//This software is copyright (C) 1991-1996, Thomas G. Lane.
//All Rights Reserved except as specified below.

#include "stdio.h"
#include "..\ijg\cdjpeg.h"

int main()
{
    //The input and output "C" file structures
    FILE* input_file;
    FILE* output_file;
```

```
char* input_filename = "input.jpg";
char* output_filename = "output.bmp";

//the number of scanlines read from each call to the
//pixel-reading IJG function (jpeg_read_scanlines)
JDIMENSION num_scanlines;

//open the input JPEG and output bitmap file.
input_file = fopen(input_filename, READ_BINARY);
output_file = fopen(output_filename, WRITE_BINARY);

//create a JPEG decompression structure on the stack
//this structure is used by IJG to store JPEG-
//specific information and decompression parameters
struct jpeg_decompress_struct cinfo;

//create a JPEG error handler structure on the stack
//IJG permits user-defined error management
struct jpeg_error_mgr jerr;
cinfo.err = jpeg_std_error(&jerr);

//initialize the JPEG decompression structure
jpeg_create_decompress(&cinfo);

//set the JPEG input as a standard file
jpeg_stdio_src(&cinfo, input_file);

//The "destination manager" is a file format manager built into
//IJG that performs conversions between IJG data and
//other file formats. In this case, we initialize the manager to
//deal with 24-bit (true color) Windows Bitmaps.
djpeg_dest_ptr dest_mgr = jinit_write_bmp(&cinfo, FALSE);
dest_mgr->output_file = output_file;

//Read the file header
//TRUE indicates we will use the default decompression parameters
jpeg_read_header(&cinfo, TRUE);

//begin decoding the JPEG file. Read image parameters,
//and header information.
jpeg_start_decompress(&cinfo);

//Start writing to the destination Windows Bitmap.
//The bitmap file header and BITMAPINFOHEADER will be
//written to the output file.
(*dest_mgr->start_output) (&cinfo, dest_mgr);

//loop over the JPEG image: read groups of scanlines
//(each read captures num_scanlines) and write these
//groups to the output bitmap
while (cinfo.output_scanline < cinfo.output_height)
{
    //read a group of scanlines from the JPEG file. IJG decides
    //the number of scanlines it needs to read (up to the
    //maximum number specified by the "dest_mgr->buffer_height"
    //parameter) and will buffer them internally.
```

```
        num_scanlines = jpeg_read_scanlines(&cinfo, dest_mgr->buffer,
            dest_mgr->buffer_height);

        //write these buffered scanlines to the destination manager.
        //the destination manager is capable of writing a variable
        //number of scanlines to the destination
        (*dest_mgr->put_pixel_rows) (&cinfo, dest_mgr, num_scanlines);
    }

    //finish writing any additional information to the Bitmap
    (*dest_mgr->finish_output) (&cinfo, dest_mgr);

    //Finish decompressing the JPEG file and
    //destroy the JPEG Decompressor
    jpeg_finish_decompress(&cinfo);
    jpeg_destroy_decompress(&cinfo);

    //close input and output files
    fclose(input_file);
    fclose(output_file);

    return 0;
}
```

The IJL can be used in two ways to accomplish the same task. We will illustrate both. The final code for each IJL implementation is found at the end of this section.

Our first approach is (like IJG) to create a temporary buffer for the decoded JPEG image and write the temporary buffer to the bitmap. We will choose a small buffer for efficiency (the buffer is set to 32 scanlines in length, because we know that this number works well for most JPEG images). The IJL does not support file types other than JPEG, so we will need to add our own bitmap header.

Looking at the first segment of the IJG code (ignoring the header and #includes), IJG requires us to open both the input and output files.

```
int main()
{
    //The input and output "C" file structures
    FILE* input_file;
    FILE* output_file;
    char* input_filename = "input.jpg";
    char* output_filename = "output.bmp";
    //the number of scanlines read from each call to the
    //pixel-reading IJG function (jpeg_read_scanlines)
    JDIMENSION num_scanlines;

    //open the input JPEG and output bitmap file.
    input_file = fopen(input_filename, READ_BINARY);
    output_file = fopen(output_filename, WRITE_BINARY);
```

```
. . .
```

The IJL is equivalent, but performs the JPEG file opening itself. We therefore can avoid it entirely:

```
int main()
{

    //input and output filenames
    FILE* output_file;
    char* input_filename = "input.jpg";
    char* output_filename = "output.bmp";

    //Open the bitmap file for output.
    output_file = fopen(output_filename, WRITE_BINARY);

    . . .
```

We then initialize the JPEG decompressor structure. For IJG, it is typically called “cinfo” and for the IJL it is called “jcropps”. IJG requires initialization of the error handler as well; the IJL includes a built-in error manager. We then tell IJG to use our input file as the source for future JPEG decompression.

```
. . .
//create a JPEG decompression structure on the stack
//this structure is used by IJG to store JPEG-
//specific information and decompression parameters
jpeg_decompress_struct cinfo;

//create a JPEG error handler structure on the stack
//IJG permits user-defined error management
jpeg_error_mgr jerr;
cinfo.err = jpeg_std_error(&jerr);

//initialize the JPEG decompression structure
jpeg_create_decompress(&cinfo);

//set the JPEG input as a standard file
jpeg_stdio_src(&cinfo, input_file);

. . .
```

The IJL uses a member of the JPEG structure to store a pointer to the filename string:

```

. . .

```

```

//create the JPEG structure on the stack

```

```

JPEG_CORE_PROPERTIES jcprops;
//initialize the JPEG structure
ijlInit(&jcprops);
//set the IJL data source as the input filename
jcprops.JPGFile = input_filename;
. . .

```

Now that the IJL has been initialized and the source specified, we need to specify our destination. Both applications need some information from the JPEG file; this information will be used to write the header (non-image data portions of the output Bitmap). IJG Does it for us:

```

. . .
//The "destination manager" is a file format manager built into
//IJG that performs conversions between IJG data and
//other file formats. In this case, we initialize the manager to
//deal with 24-bit (true color) Windows Bitmaps.
djpeg_dest_ptr dest_mgr = jinit_write_bmp(&cinfo, FALSE);
dest_mgr->output_file = output_file;

//Read the file header
//TRUE indicates we will use the default decompression parameters
jpeg_read_header(&cinfo, TRUE);

//begin decoding the JPEG file. Read image parameters,
//and header information.
jpeg_start_decompress(&cinfo);

//Start writing to the destination Windows Bitmap.
//The bitmap file header and BITMAPINFOHEADER will be
//written to the output file.
(*dest_mgr->start_output) (&cinfo, dest_mgr);
. . .

```

While the IJL requires us to manually initialize the Bitmap header:

```

. . .
//read JPEG parameters from the file
ijlRead(&jcprops, IJL_JFILE_READPARAMS);

//calculate the line offset of the output DIB.
//Windows DIBs are aligned to 4-byte line widths.
int DIBoffset = (jcprops.JPGwidth*3 + 3)/4*4;

```

```

BITMAPFILEHEADER bmfh;
.....

BITMAPINFOHEADER bmih;

bmfh.bfType = 'MB';
bmfh.bfSize = DIBOffset * jcprops.JPGHeight +
    sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER);
bmfh.bfReserved1 = 0;
bmfh.bfReserved2 = 0;
bmfh.bfOffBits = sizeof(BITMAPFILEHEADER) +
    sizeof(BITMAPINFOHEADER);

bmih.biSize = sizeof(BITMAPINFOHEADER);
bmih.biWidth = jcprops.JPGWidth;
bmih.biHeight = jcprops.JPGHeight;
bmih.biPlanes = 1;
bmih.biBitCount = 24;
bmih.biCompression = BI_RGB;
bmih.biSizeImage = 0;
bmih.biXPelsPerMeter = 1;
bmih.biYPelsPerMeter = 1;
bmih.biClrUsed = 0;
bmih.biClrImportant = 0;

unsigned long nwritten;

//write the bitmap file header to the bitmap
WriteFile(hbitmapfile, &bmfh, sizeof(bmfh), &nwritten, 0);
WriteFile(hbitmapfile, &bmih, sizeof(bmih), &nwritten, 0);
. . .

```

We are on the verge of reading actual image data from the JPEG image and writing it to the bitmap. We now encounter some of the most significant differences between the IJL and IJG.

The simplest use of IJG “owns” uncompressed image data produced by the library. In our usage, this “ownership” has been transferred to the destination manager. The fundamental decoding call in IJG is “jpeg_read_scanlines” which permits us to read data with a granularity of “groups of scanlines “. We loop over the image vertically, reading groups of scanlines and transferring them to the destination (in this case, our destination manager).

```

. . .
//loop over the JPEG image: read gropes of scanlines
//(each read captures num_scanlines) and write these
//groups to the output bitmap
while (cinfo.output_scanline < cinfo.output_height)
{
    //read a group of scanlines from the JPEG file. IJG decides
    //the number of scanlines it needs to read (up to the

```

```

        //parameter) and will buffer them internally.
        num_scanlines = jpeg_read_scanlines(&cinfo, dest_mgr->buffer,
        dest_mgr->buffer_height);

        //write these buffered scanlines to the destination manager.
        //the destination manager is capable of writing a variable
        //number of scanlines to the destination
        (*dest_mgr->put_pixel_rows) (&cinfo, dest_mgr, num_scanlines);
    }
    . . .

```

Without a destination manager of its own, the IJL demands that the application own the output buffer. The IJL will then be forced to allocate its own image buffer. After creating the buffer, we pass information about its size to the IJL (by setting the `jcprops.DIBxxx` parameters). The IJL gives us pixel-granularity on decoding the JPEG image (we can request, with a Rectangle-of-Interest, as little as one pixel at a time from the JPEG image) but it is significantly faster when requesting 32-pixel aligned regions for most images. As we will need to write entire scanlines to the Bitmap anyway, we choose to follow the IJG plan and create a buffer as wide as the image and 32 pixels high.

We then loop over the image, reading 32-pixel high chunks of data and writing them to the bitmap.

Note that our bitmap is “bottom up” while the JPEG is “top down”; this requires us to use a negative `DIBHeight` specification. We also encounter a special case for the last (assumed partial) 32-pixel block of the DIB.

```

    . . .
    //create a temporary buffer big enough to hold a 32-line chunk of
    //the bitmap. This buffer will be written to by IJL; we will
    //then write this buffer to the bitmap.
    unsigned char* dibBuffer = new unsigned char [DIBOffset * 32];

    //set up the buffer specification for the JPEG decoder
    jcprops.DIBBytes = dibBuffer;
    jcprops.DIBWidth = jcprops.JPGWidth;
    jcprops.DIBColor = IJL_BGR;
    jcprops.DIBChannels = 3;
    jcprops.DIBPadBytes = IJL_DIB_PAD_BYTES(
        jcprops.DIBWidth, jcprops.DIBChannels);
    jcprops.DIBHeight = -32;

    for (int i = 0; i < (int)jcprops.JPGHeight; i += 32)
    {
        //read from a small region of the JPEG image
        jcprops.jprops.roi.left = 0;
        jcprops.jprops.roi.right = jcprops.JPGWidth;
    }

```

```
jcprops.jcprops.roi.bottom = i, + 32;
```

```
    //read data from the JPEG image into the bitmap
    ijlRead(&jcprops, IJL_JFILE_READENTROPY);

    //write data from the temporary buffer to the bitmap file.
    fwrite(dibBuffer, 1,
           DIBOffset * min(32, jcprops.JPGHeight - i),
           output_file);
}
. . .
```

IJG does not handle this special case in the main decoding loop, so we need to include an additional instruction to write the last few lines to the bitmap:

```
. . .
//finish writing any additional information to the Bitmap
(*dest_mgr->finish_output) (&cinfo, dest_mgr);
. . .
```

We will then free IJG and close the input and output files:

```
. . .
//Finish decompressing the JPEG file and
//destroy the JPEG Decompressor
jpeg_finish_decompress(&cinfo);
jpeg_destroy_decompress(&cinfo);

//close input and output files
fclose(input_file);
fclose(output_file);

return 0;
}
```

and likewise for the IJL:

```
. . .
//clean up and destroy the JPEG Decompressor
ijlFree(&jcprops);
```

```
    delete [] dibBuffer;

    //close the output file

fclose(output_file);

    return 0;
}
```

The second technique we may choose to exploit uses the Microsoft Win32 to create a memory-mapped Bitmap. It has the advantages of avoiding an output buffer (though Windows may create an intermediate buffer internally) and allowing the OS to best determine how to buffer data for optimal use of the destination file. The technique is in general not as extensible to other decoding requirements (for example, writing to a DIB in memory or a 24-bit display surface). The example for this code is found in the appendix.

```
#include "windows.h"
#include "ijl.h"

int main()
{

    // input and output filenames
    FILE* output_file;
    char* input_filename = "input.jpg";
    char* output_filename = "output.bmp";

    // Open the bitmap file for output.
    output_file = fopen(output_filename, "wb+");

    // create the JPEG structure on the stack
    JPEG_CORE_PROPERTIES jcprops;
    // initialize the JPEG structure
    ijlInit(&jcprops);
    // set the IJL data source as the input filename
    jcprops.JPGFile = input_filename;
    // read JPEG parameters from the file
    ijlRead(&jcprops, IJL_JFILE_READPARAMS);

    // calculate the line offset of the output DIB.
    // Windows DIBs are aligned to 4-byte line widths.
    int DIBoffset = (jcprops.JPGWidth*3 + 3)/4*4;

    // write the output bitmap header
    BITMAPFILEHEADER bmfh;
    BITMAPINFOHEADER bmih;

    bmfh.bfType      = 'MB';
```

```

        sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER);
    bmfh.bfReserved1 = 0;
    bmfh.bfReserved2 = 0;
    bmfh.bfOffBits   = sizeof(BITMAPFILEHEADER) +
        sizeof(BITMAPINFOHEADER);

```

```

    bmih.biSize          = sizeof(BITMAPINFOHEADER);
    bmih.biWidth        = jcprops.JPGWidth;
    bmih.biHeight       = jcprops.JPGHeight;
    bmih.biPlanes       = 1;
    bmih.biBitCount     = 24;
    bmih.biCompression = BI_RGB;
    bmih.biSizeImage    = 0;
    bmih.biXPelsPerMeter = 1;
    bmih.biYPelsPerMeter = 1;
    bmih.biClrUsed      = 0;
    bmih.biClrImportant = 0;

    unsigned long nwritten;

    // write the bitmap file header to the bitmap
    WriteFile(hbitmapfile, &bmfh, sizeof(bmfh), &nwritten, 0);
    WriteFile(hbitmapfile, &bmih, sizeof(bmih), &nwritten, 0);

    // create a temporary buffer big enough to hold a 32-line chunk of
    // the bitmap. This buffer will be written to by IJL; we will
    // then write this buffer to the bitmap.
    unsigned char* dibBuffer = new unsigned char [DIBOffset * 32];

    // set up the buffer specification for the JPEG decoder
    jcprops.DIBBytes      = dibBuffer;
    jcprops.DIBWidth     = jcprops.JPGWidth;
    jcprops.DIBColor     = IJL_BGR;
    jcprops.DIBChannels  = 3;
    jcprops.DIBPadBytes  = DIBOffset - 3*jcprops.JPGWidth;
    jcprops.DIBHeight    = -32;

    for (int i = 0; i < (int)jcprops.JPGHeight; i += 32)
    {
        // read from a small region of the JPEG image
        jcprops.jprops.roi.left   = 0;
        jcprops.jprops.roi.right  = jcprops.JPGWidth;
        jcprops.jprops.roi.top    = i;
        jcprops.jprops.roi.bottom = i + 32;

        // read data from the JPEG image into the bitmap
        ijlRead(&jcprops, IJL_JFILE_READENTROPY);

        // write data from the temporary buffer to the bitmap file.
        fwrite(dibBuffer, 1,
            DIBOffset * min(32, jcprops.JPGHeight - i),
            output_file);
    }

```

```
    // clean up and destroy the JPEG Decompressor
    ijlFree(&jcprops);

    // release the buffer

delete [] dibBuffer;

    // close the output file
    fclose(output_file);

    return 0;
}
```

2.2 Matching the IJG Sample Application “cjpeg.exe”

Like “djpeg.exe”, “cjpeg.exe” is a DOS application that permits command-line configuration; except this time in the JPEG encoder.

Like the decoder, there are a couple of ways to map the IJL to “cjpeg.c”'s functionality. In the first we copy a bitmap into a temporary buffer and use the IJL to encode the buffer to a JPEG file. In Appendix B, we illustrate a method to accomplish the same end without a temporary buffer – by using Windows memory-mapped files.

The code for a simplified version of the IJG source “cjpeg.c” is given below.

```
// Note: This code is based on a file "djpeg.c" from the Independent
// JPEG Group (IJG) JPEG codec Version 6a. Please see the original
// IJG README file for legal information relating to code distribution
// and attribution

// IJG Copyright:
// This software is copyright (C) 1991-1996, Thomas G. Lane.
// All Rights Reserved except as specified below.

#include <stdio.h>
#include "..\ijg\cdjpeg.h"

int main()
{
    // open input and output files
    FILE* input_file;
    FILE* output_file;
    char* input_filename = "input.bmp";
    char* output_filename = "output.jpg";

    input_file = fopen(input_filename, READ_BINARY);
    output_file = fopen(output_filename, WRITE_BINARY);
```

```
jpeg_compress_struct cinfo;

// Initialize the JPEG compression object with default error
// handling.
jpeg_error_mgr jerr;
cinfo.err = jpeg_std_error(&jerr);
jpeg_create_compress(&cinfo);

// set the default compression parameters.  IJG's defaults produce

// a "standard" JFIF image with 4:1:1 subsampling in a YUV color
// space at average quality.
cinfo.in_color_space = JCS_RGB;
jpeg_set_defaults(&cinfo);

// Set the input format for our source manager as a Windows
// Bitmap file
cjpeg_source_ptr src_mgr = jinit_read_bmp(&cinfo);
src_mgr->input_file = input_file;

// Read the input file header to obtain file size & colorspace.
(*src_mgr->start_input) (&cinfo, src_mgr);

// Now that we know input colorspace, fix colorspace-dependent
// defaults
jpeg_default_colorspace(&cinfo);

// Specify data destination for compression
jpeg_stdio_dest(&cinfo, output_file);

// Start compressor
jpeg_start_compress(&cinfo, TRUE);

// Process data
while (cinfo.next_scanline < cinfo.image_height)
{
    // read data from the source into the source buffer
    // num_scanlines will be read.
    int num_scanlines = (*src_mgr->get_pixel_rows) (
        &cinfo,
        src_mgr);

    // write num_scanlines scanlines of data from the source
    // manager input buffer (src_mgr->buffer) to the JPEG file.
    jpeg_write_scanlines
        (&cinfo, src_mgr->buffer, num_scanlines);
}

// Finish reading data from the Bitmap and writing it to
// the JPEG image.
(*src_mgr->finish_input) (&cinfo, src_mgr);
jpeg_finish_compress(&cinfo);

// Clean up the JPEG Compressor structure.
jpeg_destroy_compress(&cinfo);
```

```
    fclose(input_file);
    fclose(output_file);

    return 0;
}
```

Like the “djpeg.c” case above, we will map the various segments of this program into equivalent portions in the IJL.

Like in “djpeg.c” we need to initialize our I/O files:

```
int main()
{
    // open input and output files
    FILE* input_file;
    FILE* output_file;
    char* input_filename = "input.bmp";
    char* output_filename = "output.jpg";

    input_file = fopen(input_filename, READ_BINARY);
    output_file = fopen(output_filename, WRITE_BINARY);
    . . .
```

The IJL handles JPEG file opening and closing directly – we do not need to explicitly open the output file:

```
int main()
{
    // open input and output files
    FILE* input_file;
    char* input_filename = "input.bmp";
    char* output_filename = "output.jpg";
    input_file = fopen(input_filename, "rb");
    . . .
```

IJG next needs to initialize its JPEG compressor structure and set the encoder to create a default JPEG image. This image consists of a YCbCr 4:1:1 subsampled JPEG image with an average (75 out of 100) quality level.

```
. . .
// create the JPEG compression structure.
jpeg_compress_struct cinfo;
```

```
// handling.
jpeg_error_mgr jerr;
cinfo.err = jpeg_std_error(&jerr);
jpeg_create_compress(&cinfo);

// set the default compression parameters.  IJG's defaults produce
// a "standard" JFIF image with 4:1:1 subsampling in a YUV color
// space at average quality.

cinfo.in_color_space = JCS_RGB;
jpeg_set_defaults(&cinfo);
. . .
```

And in the IJL:

```
. . .
// create an IJL compressor; initialize with default
// parameters
JPEG_CORE_PROPERTIES jcprops;
ijlInit(&jcprops);
. . .
```

Initializing data from the bitmap is simple in IJG, which has a source module that hides the bitmap parsing from the user. This “source manager” is not present in the IJL, therefore it is necessary to configure IJG to read from the source image:

```
. . .
// Set the input format for our source manager as a Windows
// Bitmap file
cjpeg_source_ptr src_mgr = jinit_read_bmp(&cinfo);
src_mgr->input_file = input_file;

// Read the input file header to obtain file size & colorspace.
(*src_mgr->start_input) (&cinfo, src_mgr);
. . .
```

The IJL needs to:

```
. . .
// read the bitmap headers
// these headers contain the bitmap parameters
// (height, width, etc)
```

```

    BITMAPINFOHEADER bmih;

    fread(&bmfh, 1, sizeof(BITMAPFILEHEADER), input_file);
    fread(&bmih, 1, sizeof(BITMAPINFOHEADER), input_file);

    // determine the width of one line of the input bitmap in bytes

    // Windows Bitmaps are padded to make each line width an even

    // multiple of 32 bits.
    long DIBLineSize = (bmih.biWidth * 3)/4*4;

    // initialize the compression parameters describing the kind of
    // Bitmap we are encoding from.
    jcprops.DIBChannels = 3;
    jcprops.DIBColor    = IJL_BGR;
    jcprops.DIBHeight  = bmih.biHeight;
    jcprops.DIBWidth   = bmih.biWidth;
    jcprops.DIBPadBytes = IJL_DIB_PAD_BYTES(
jcprops.DIBWidth, jcprops.DIBChannels);
    . . .

```

We can now finish initializing the JPEG parameters prior to compression:

```

    . . .
    // Now that we know input colorspace, fix colorspace-dependent
    // defaults
    jpeg_default_colorspace(&cinfo);

    // Specify data destination for compression
    jpeg_stdio_dest(&cinfo, output_file);

    // Start compressor
    jpeg_start_compress(&cinfo, TRUE);
    . . .

```

and in the IJL:

```

    . . .
    // initialize the compression parameters describing the kind of
    // JPEG to create. We are using default values for image quality,
    // JPEG color space, etc.
    jcprops.JPGHeight = bmih.biHeight;
    jcprops.JPGWidth  = bmih.biWidth;

    // Set the JPEG compressor to write to the "output.jpg" file
    jcprops.JPGFile = output_filename;

```

Compressing the actual image data in IJG follows the same technique (but reversed) as decompressing; we loop over the scanlines in the image, read a variable number of scanlines from the source manager into an internal buffer, and write these scanlines to the output.

```

. . .
// Process data
while (cinfo.next_scanline < cinfo.image_height)
{
    // read data from the source into the source buffer
    // num_scanlines will be read.
    int num_scanlines = (*src_mgr->get_pixel_rows) (
        &cinfo,
        src_mgr);

    // write num_scanlines scanlines of data from the source
    // manager input buffer (src_mgr->buffer) to the JPEG file.
    jpeg_write_scanlines
        (&cinfo, src_mgr->buffer, num_scanlines);
}

// Finish reading data from the Bitmap and writing it to
// the JPEG image.
(*src_mgr->finish_input) (&cinfo, src_mgr);
jpeg_finish_compress(&cinfo);
. . .

```

The IJL does not currently support scanline-based encoding – we must encode the whole file in one step. This requires a temporary buffer big enough to hold the entire JPEG image. This buffer is set as the source of the image data inside of the IJL structure, before the single call to `ijlWrite` is made.

```

// allocate a DIB large enough to hold the entire bitmap.
// this DIB will be passed to the encoder as the actual input.
. . .
// Note that we could use other, more efficient techniques
// to map the bitmap file into an address range IJL can interpret;
// Windows memory-mapped files would be a good solution.
unsigned char* DIBBuffer = new unsigned char [DIBLineSize *
    bmih.biHeight];

// read the bitmap into the memory buffer.
// In some bitmap files the data bits will not immediately
// follow the end of the BITMAPINFOHEADER. I am assuming for
// this exercise that the bitmap is a "traditional" bottom-up
// 24-bit Windows DIB where this condition holds.

```

```
// set the JPEG DIB source to the newly allocated temporary
// buffer.
jcprops.DIBBytes = DIBBuffer;

// write the entire JPEG image. This creates a JFIF file.
ijlWrite(&jcprops, IJL_JFILE_WRITEWHOLEIMAGE);
. . .
```

Cleanup is trivial for both:

```
. . .
// Clean up the JPEG Compressor structure.
jpeg_destroy_compress(&cinfo);

// Close the input and output files
fclose(input_file);
fclose(output_file);

return 0;
}
```

and in the IJL:

```
. . .
// release the IJL
ijlFree(&jcprops);

// delete the temporary DIB buffer
delete [] DIBBuffer;

// close the input Bitmap;
fclose(input_file);

return 0;
}
```

The total code for the “djpeg.c”-equivalent JPEG decoder is:

```
#include "ijl.h"
#include <stdio.h>
#include "wingdi.h"
```

```
int main()
{
    // open input and output files
    FILE* input_file;
    char* input_filename = "input.bmp";
    char* output_filename = "output.jpg";
    input_file = fopen(input_filename, "rb");

    // create an IJL compressor; initialize with default

    // parameters
    JPEG_CORE_PROPERTIES jcprops;
    ijInit(&jcprops);

    // read the bitmap headers
    // these headers contain the bitmap parameters
    // (height, width, etc)
    BITMAPFILEHEADER bmfh;
    BITMAPINFOHEADER bmih;

    fread(&bmfh, 1, sizeof(BITMAPFILEHEADER), input_file);
    fread(&bmih, 1, sizeof(BITMAPINFOHEADER), input_file);

    // determine the width of one line of the input bitmap in bytes
    // Windows Bitmaps are padded to make each line width an even
    // multiple of 32 bits.
    long DIBLineSize = (bmih.biWidth * 3)/4*4;

    // initialize the compression parameters describing the kind of
    // Bitmap we are encoding from.
    jcprops.DIBChannels = 3;
    jcprops.DIBColor    = IJL_BGR;
    jcprops.DIBHeight   = bmih.biHeight;
    jcprops.DIBWidth    = bmih.biWidth;
    jcprops.DIBPadBytes = IJL_DIB_PAD_BYTES(
jcprops.DIBWidth, jcprops.DIBChannels);

    // initialize the compression parameters describing the kind of
    // JPEG to create. We are using default values for image quality,
    // JPEG color space, etc.
    jcprops.JPGHeight = bmih.biHeight;
    jcprops.JPGWidth  = bmih.biWidth;

    // Set the JPEG compressor to write to the "output.jpg" file
    jcprops.JPGFile = output_filename;

    // allocate a DIB large enough to hold the entire bitmap.
    // this DIB will be passed to the encoder as the actual input.
    // Note that we could use other, more efficient techniques
    // to map the bitmap file into an address range IJL can interpret;
    // Windows memory-mapped files would be a good solution.
    unsigned char* DIBBuffer = new unsigned char [DIBLineSize *
        bmih.biHeight];
```

```
// read the bitmap into the memory buffer.
// In some bitmap files the data bits will not immediately
// follow the end of the BITMAPINFOHEADER. I am assuming for
// this exercise that the bitmap is a "traditional" bottom-up
// 24-bit Windows DIB where this condition holds.
fread(DIBBuffer, 1, DIBLineSize * bmih.biHeight, input_file);

// set the JPEG DIB source to the newly allocated temporary
// buffer.
jcprops.DIBBytes = DIBBuffer;
```

```
// write the entire JPEG image. This creates a JFIF file.
ijlWrite(&jcprops, IJL_JFILE_WRITEWHOLEIMAGE);

// release the IJL
ijlFree(&jcprops);

// delete the temporary DIB buffer
delete [] DIBBuffer;

// close the input Bitmap;
fclose(input_file);

return 0;
}
```

2.3 Quality differences between IJL and IJG encoder quality settings

The IJL produces equivalent quality and compressed image size to IJG given an identical "quality level" (0-100) passed to the encoder at run time.

3. Appendix A: A Memory-Mapped Bitmap implementation of a JPEG Decoder.

This section illustrates the use of the IJL to decode a JPEG image directly to a bitmap file (using Windows Memory Mapped files).

```
// Equivalent IJL based application
#include "stdio.h"
#include "ijl.h"

int main()
{
    // input and output filenames
    char* input_filename = "input.jpg";
    char* output_filename = "output.bmp";

    // declare the file handle
    HANDLE hbitmapfile;
    // Open the bitmap file for output.
    hbitmapfile = CreateFile(
        output_filename,
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE,
        0,
        CREATE_ALWAYS,
        0, 0);

    // create the JPEG structure on the stack
    JPEG_CORE_PROPERTIES jcprops;
    // initialize the JPEG structure
    ijInit(&jcprops);
    // set the IJL data source as the input filename
    jcprops.JPGFile = input_filename;
    // read JPEG parameters from the file
    ijRead(&jcprops, IJL_JFILE_READPARAMS);
    // calculate the line offset of the output DIB.
    // Windows DIBs are aligned to 4-byte line widths.
    int DIBOffset = (jcprops.JPGWidth*3 + 3)/4*4;
    // resize the output bitmap file to the actual size the
    // bitmap will represent.
    SetFilePointer(
        hbitmapfile,
        DIBOffset * jcprops.JPGHeight + sizeof(BITMAPFILEHEADER)
        + sizeof(BITMAPINFOHEADER),
        0,
        FILE_BEGIN);

    SetEndOfFile(hbitmapfile);

    // map the disk file into an addressable memory region.
    HANDLE hFileMappingObject;

    hFileMappingObject = CreateFileMapping(
```

```
        hbitmapfile,  
        0,  
        PAGE_READWRITE,  
        0, 0,  
        0);  
  
    // the memory-mapped file will reside at address  
    // "bitmapptr"  
    unsigned char* bitmapptr;  
    bitmapptr = (unsigned char*) MapViewOfFile(  
        hFileMappingObject,  
        FILE_MAP_WRITE,  
        0, 0, 0);  
  
    // write the output bitmap header  
    BITMAPFILEHEADER *bmfh = (BITMAPFILEHEADER *)bitmapptr;  
    BITMAPINFOHEADER *bmih = (BITMAPINFOHEADER *) (bitmapptr  
        + sizeof(BITMAPFILEHEADER));  
  
    bmfh->bfType = 'MB';  
    bmfh->bfSize = DIBOffset * jcprops.JPGHeight  
        + sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER);  
    bmfh->bfReserved1 = 0;  
    bmfh->bfReserved2 = 0;  
    bmfh->bfOffBits = sizeof(BITMAPFILEHEADER)  
        + sizeof(BITMAPINFOHEADER);  
  
    bmih->biSize = sizeof(BITMAPINFOHEADER);  
    bmih->biWidth = jcprops.JPGWidth;  
    bmih->biHeight = jcprops.JPGHeight;  
    bmih->biPlanes = 1;  
    bmih->biBitCount = 24;  
    bmih->biCompression = BI_RGB;  
    bmih->biSizeImage = 0;  
    bmih->biXPelsPerMeter = 1;  
    bmih->biYPelsPerMeter = 1;  
    bmih->biClrUsed = 0;  
    bmih->biClrImportant = 0;  
  
    // set up the DIB specification for the JPEG decoder  
    jcprops.DIBBytes = bitmapptr + sizeof(BITMAPFILEHEADER)  
        + sizeof(BITMAPINFOHEADER);  
    jcprops.DIBWidth = jcprops.JPGWidth;  
    jcprops.DIBHeight = -(int)jcprops.JPGHeight;  
    jcprops.DIBColor = IJL_BGR;  
    jcprops.DIBChannels = 3;  
    jcprops.DIBPadBytes = IJL_DIB_PAD_BYTES(  
        jcprops.DIBWidth, jcprops.DIBChannels);  
  
    // read data from the JPEG image into the bitmap  
    ijLRead(&jcprops, IJL_JFILE_READWHOLEIMAGE);  
  
    // clean up and destroy the JPEG Decompressor  
    ijLFree(&jcprops);
```

```
    // close the output file
    UnmapViewOfFile(bitmapptr);
    CloseHandle(hFileMappingObject);
    CloseHandle(hbitmapfile);

    return 0;
}
```

4. Appendix B: A Memory-Mapped Bitmap implementation of a JPEG Encoder.

This section illustrates the use of the IJL to encode a Windows Bitmap to a JPEG image using Windows Memory Mapped files.

```
#include "ijl.h"
#include "windows.h"

int main()
{
    // open input and output files
    char* input_filename = "input.bmp";
    char* output_filename = "output.jpg";

    // create an IJL compressor; initialize with default
    // parameters
    JPEG_CORE_PROPERTIES jcprops;
    ij1Init(&jcprops);

    // declare the file handle
    HANDLE hbitmapfile;
    // Open the bitmap file for output.
    hbitmapfile = CreateFile(
        input_filename,
        GENERIC_READ,
        0,
        0,
        OPEN_EXISTING,
        0, 0);

    // map the disk file into an addressable memory region.
    HANDLE hFileMappingObject;

    hFileMappingObject = CreateFileMapping(
        hbitmapfile,
        0,
        PAGE_READONLY,
        0, 0,
        0);

    // the memory-mapped file will reside at address
    // "bitmapptr"
    unsigned char* bitmapptr;
    bitmapptr = (unsigned char*) MapViewOfFile(
        hFileMappingObject,
        FILE_MAP_READ,
        0, 0, 0);

    // read the input bitmap header
```

```
BITMAPFILEHEADER* bmfh = (BITMAPFILEHEADER*)bitmapptr;
BITMAPINFOHEADER* bmih = (BITMAPINFOHEADER*)(bitmapptr
    + sizeof(BITMAPFILEHEADER));

// determine the width of one line of the input bitmap in bytes
// Windows Bitmaps are padded to make each line width an even
// multiple of 32 bits.
long DIBLineSize = (bmih->biWidth * 3)/4*4;

// initialize the compression parameters describing the kind of
// Bitmap we are encoding from.
jcprops.DIBChannels = 3;
jcprops.DIBColor = IJL_BGR;
jcprops.DIBHeight = bmih->biHeight;
jcprops.DIBWidth = bmih->biWidth;
jcprops.DIBPadBytes = IJL_DIB_PAD_BYTES(
    jcprops.DIBWidth, jcprops.DIBChannels);

// initialize the compression parameters describing the kind of
// JPEG to create. We are using default values for image quality,
// JPEG color space, etc.
jcprops.JPGHeight = bmih->biHeight;
jcprops.JPGWidth = bmih->biWidth;

// Set the JPEG compressor to write to the "output.jpg" file
jcprops.JPGFile = output_filename;

// set the JPEG DIB source to the newly allocated temporary
// buffer.
jcprops.DIBBytes = bitmapptr
    + sizeof(BITMAPINFOHEADER) + sizeof(BITMAPFILEHEADER);

// write the entire JPEG image. This creates a JFIF file.
ijlWrite(&jcprops, IJL_JFILE_WRITEWHOLEIMAGE);

// release the IJL
ijlFree(&jcprops);

// Release file mapping object and close
// output file
UnmapViewOfFile(bitmapptr);
CloseHandle(hFileMappingObject);
CloseHandle(hbitmapfile);

return 0;
}
```