Intel® JPEG Library

Developer's Guide

Issued in U.S.A.
Document number 726916-006

Intel® JPEG Library Developer's Guide

Document number: 726916-006

World Wide Web: http://developer.intel.com

Revision	Revision History	Date
-001	First release.	09/98
-002	Added the functions ijlGetLibVersion and ijlErrorStr	01/99
-003	Added new code examples	07/99
-004	Documents the Intel® JPEG Library version 1.5	07/00
-005	New operations with user-defined buffers have been added	10/00
-006	Raw DCT data processing and support of pixel-interleaved YCbCr422 format have been added.	04/01

Contents

Figures

Tables

Examples

Overview

1

This Developer's Guide describes the design and implementation of the Intel® JPEG Library (IJL). Please use this guide in conjunction with the source code for the Sample Application and with the other IJL documentation as a learning tool to familiarize yourself with the use of the IJL.

This guide assumes that the reader has a working knowledge of the software development process and the C/C++ programming language. Some familiarity with digital imaging, software development for the Microsoft* Windows* 95, 98 operating systems, and the Microsoft Foundation Classes application framework may also be useful.

A note to the reader, the following appendices are located at the end of this document for reference: *Appendix A - Glossary of Terms*, and *Appendix B - Data Structure and Type Definitions* (which provides additional information on IJL data structures, type definitions, and error codes).

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® processors-based systems 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.

Minimum Requirements

- The IJL requires the presence of the Microsoft Windows 95, 98 or Windows NT* operating system, and uses the Win32* application programming interface (API).
- The IJL was designed to run on at least an Intel® Pentium® processor.
- A 32-bit compiler is required to create a 32-bit IJL application.
- Since the IJL is a Dynamic Link Library (DLL), the programming language used must be able to produce an application capable of calling functions contained within a Win32 DLL.

What's New in IJL

The IJL version 1.5 supports the following new features:

- Encoding of progressive JPEG images.
- New DCT algorithm of higher accuracy, derived from Intel® Integrated Performance Primitives for Intel® architecture.
- New sampling algorithm with triangular filter, `IJL_TRIANGLE_FILTER`, which gives better quality results.
- New input data format for encoding and output format for decoding, which is 4:2:2 subsampled pixel-interleaved `IJL_YCBCR` format with data sequence set as a Y0-Cb0-Y1-Cr0-Y2-Cb1-Y3-Cr1-... .
- Support of new instructions for Pentium 4 processor.
- Writing and reading of JPEG segment that contains comments.
- Starting from version 1.5, the library can write the JPEG tables detected while decoding, into a user-defined buffer .

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 visit the IJL support page at http://support.intel.com/support/performancetools/libraries/ijl/index.htm

Programming Considerations 2

There are four components necessary for creating an IJL application:

1. The IJL dynamic link library (**IJL15.DLL**),
2. The IJL static link library (**IJL15L.LIB**),
3. The IJL import library (**IJL15.LIB**), and
4. The IJL header file (**IJL.H**).

Dynamic Link Library

The dynamic link library (DLL) contains the IJL functions called by your application during execution. Digits after the name indicate the current library version.

Static Link Library

The static link library (**IJL15L.LIB**) contains the IJL functions called by your application during execution. Digits after the name indicate the current library version.

Import Library

The import library is linked to your application at compilation time and relates the IJL function calls to actual entry points in the DLL.

Header File

The header file contains the IJL function declarations and provides data structure definitions, data type definitions, and error codes.

Steps for Creating an IJL Application

1. Write your program with the IJL function calls. Use the IJL functions just as if they were defined in your program.
2. Include the IJL header file, **`IJL.H`**, in each source module that calls an IJL function.
3. Add the IJL import library **`IJL15.LIB`**, or static library **`IJL15L.LIB`** to your project's list of link libraries.
4. Compile and link your application as you would normally do to create a Win32 application.

Architecture Description 3

The current JPEG standard (ISO DIS 10918-1) has 44 possible JPEG image compression techniques, many of which are application-specific and not used by the majority of the JPEG decoders. Similarly, the IJL supports only a subset of the possible compression techniques.

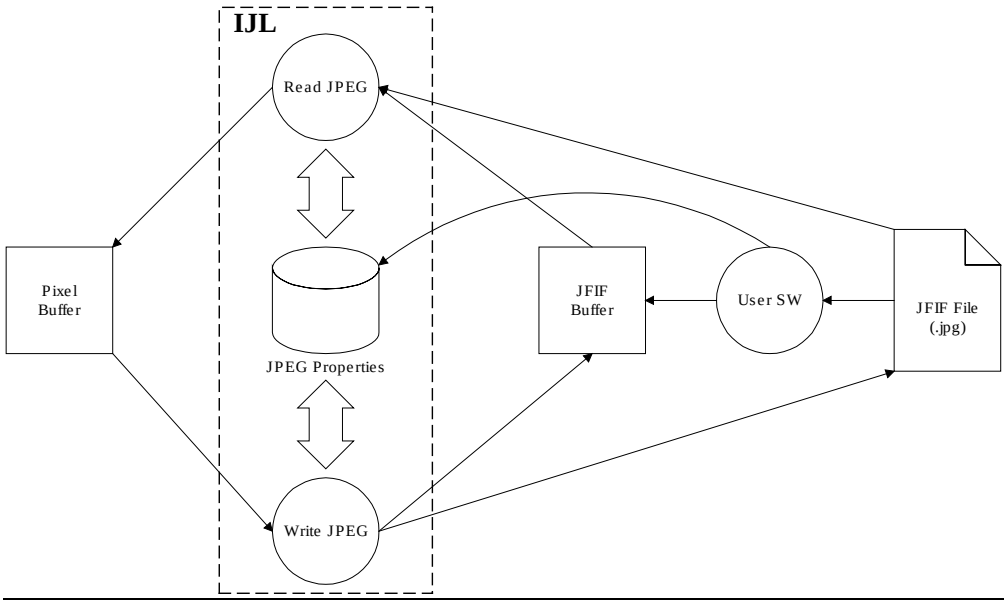
Today, the most commonly used JPEG modes are the sequential DCT-based Baseline and Extended Baseline modes. Both of these are fully supported in the IJL for JPEG encoding and decoding. The IJL version 1.5 supports also Progressive modes for JPEG encoding and decoding. There is currently no provision for restart intervals in Progressive encoding mode.

Supported I/O Data Structures

The IJL architecture (see Figure 3-1) performs basic input from, and output to, these data structures:

1. A general pixel buffer in memory.
2. A standard I/O file that contains a JPEG bit stream.
3. A memory buffer that contains a JPEG bit stream.

Figure 3-1 Top-Level Architecture of the Intel® JPEG Library



Supported Data Formats

Additionally, the IJL supports the following data formats:

- Top-down or bottom-up pixel buffers.
- Pixel buffers with user-defined end-of-line padding.
- Access to a rectangle-of-interest within a general pixel buffer.
- Decoding from a rectangle-of-interest within a larger JPEG image.
- JPEG File Interchange Format (JFIF) and abbreviated format encoding and decoding. IJL provides decoding of JFIF files compliant with JFIF specification version 1.02. Encoding is done as per JFIF version 1.01. IJL also supports decoding of embedded uncompressed thumbnails stored using 1 or 3 bytes/pixel as compliant with JFIF specification version 1.02. Thumbnails compressed using JPEG are not supported at this time.

Data (sample) values must be 8-bits precision per color channel.

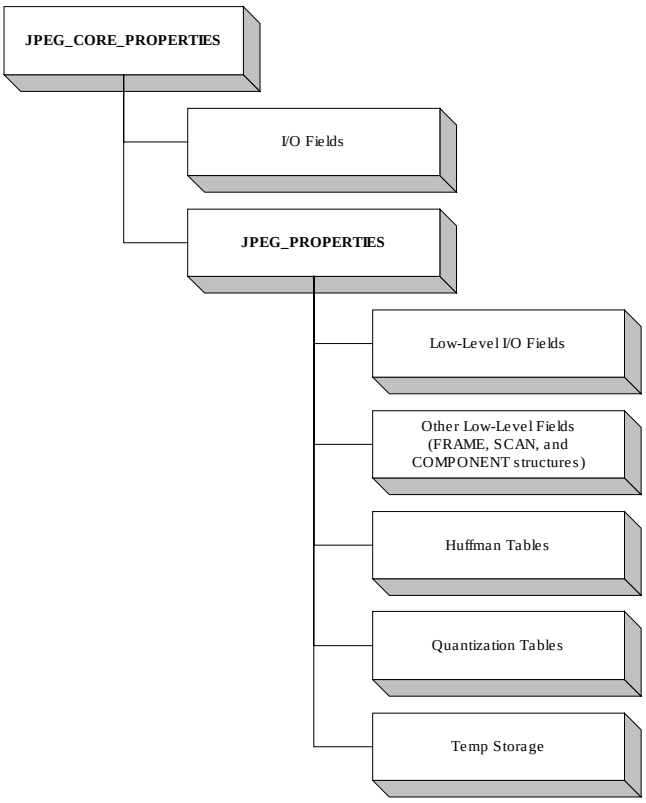
JPEG Properties Data Storage

The IJL's "JPEG Properties" data storage contains global and image-specific JPEG information. Control structures within this storage determine I/O specific processing options, such as subsampling and color conversion requirements.
The IJL uses the `JPEG_CORE_PROPERTIES` data structure for storing the JPEG properties data. This structure can be described as having two separate parts. The first part consists of a set of fields encapsulating common library parameters, and the other part consists of a low-level embedded structure (see Figure 3-2 and/or *Appendix B - Data Structure and Type Definitions*).

Users must follow two main rules about `JPEG_CORE_PROPERTIES`:

1. The user must always provide (allocate) the `JPEG_CORE_PROPERTIES` data structure.
2. The same `JPEG_CORE_PROPERTIES` data structure may be reused for a series of JPEG encodings and/or decodings when initialized and cleaned up properly.

Figure 3-2 The Intel® JPEG Library Main Data Structure



`JPEG_PROPERTIES` is the low-level data structure and it contains a copy of each of the fields found inside of the `JPEG_CORE_PROPERTIES` high-level data structure plus some additional fields. The IJL uses `JPEG_PROPERTIES`

internally, not JPEG_CORE_PROPERTIES, so this structure isolates the internal variables from the external.

For advanced users, the JPEG_PROPERTIES data structure may be used for extended interface behavior. For example, the user may want to write user-defined Huffman tables and/or quantization tables directly to JPEG_PROPERTIES to override the default tables (see *Chapter 7, Advanced IJL Features* for more information).

Default values for the fields in both the JPEG_CORE_PROPERTIES and JPEG_PROPERTIES data structures are fully documented as in-line comments inside of the header file **IJL.H**.

Multi-Threading Support

The JPEG_CORE_PROPERTIES data structure was designed to be local to a single thread. There is no parameter "locking" that will allow multiple threads to access the same JPEG_CORE_PROPERTIES structure. However, all implementation details of the IJL allow multiple JPEG_CORE_PROPERTIES storages and code access by multiple threads.

Interface Specifications

4

The IJL provides a simple C function interface (see Figure 4-1). It was modeled on a simple read/write stack built around the `JPEG_CORE_PROPERTIES` data structure.

There are functions to initialize and release the storage used inside of `JPEG_CORE_PROPERTIES`. Also provided are functions designed to transact data and/or parameters to, or from, the IJL.

IJL function calls return a descriptive error code upon a failure; otherwise, a positive success code (`IJL_OK`, `IJL_INTERRUPT_OK`, `IJL_ROI_OK`) is returned. See *Appendix B - Data Structure and Type Definitions* for further details. To convert an error code to a string with the textual description of the error, use the function **`ijlErrorStr()`**.

Finally, the function **`ijlGetLibVersion()`** returns the version number and other information about the library.
Note that both the **`ijlErrorStr()`** and **`ijlGetLibVersion()`** functions return a pointer to a static variable, so the application has no need to free the memory referenced by these pointers.

Figure 4-1 The Intel® JPEG Library Application Programming Interface

```
// Initialize the IJL.
IJLERR ijlInit (JPEG_CORE_PROPERTIES *jcprops);

// Clean up the IJL.
IJLERR ijlFree (JPEG_CORE_PROPERTIES *jcprops);

// Use the IJL to read data from a buffer or a file.
IJLERR ijlRead (JPEG_CORE_PROPERTIES *jcprops, IJLIOTYPE
iotype);
```

continued

Figure 4-1 The Intel JPEG Library Application Programming Interface (continued)

```
// Use the IJL to write data into a buffer or a file.
IJLERR ijlWrite (JPEG_CORE_PROPERTIES *jcprops, IJLIOTYPE
iotype);

// Return the version number of the IJL.
const IJLibVersion* ijlGetLibVersion( );

// Return a pointer to a string with error description.
const char* ijlErrorStr(IJLERR code);
```

Inside the Library 5

This section describes the design and implementation of common features of the IJL, as well as providing some working examples.

Initialization

The IJL must be initialized before it can be used by an application. This occurs in the **`ijlInit( )`** function. This function should only be called once per each allocation of a `JPEG_CORE_PROPERTIES` data structure.

In the event that an application wants to make multiple calls to either the encode or decode functions, the application should include a call to **`ijlInit( )`** before either of the functions is invoked, and more precisely the initialization needs to take place between each individual call to the encode or decode functions. This allows the `JPEG_CORE_PROPERTIES` data structure to reset. Additionally, there must be a one-to-one correlation between each initialization call and its counterpart **`ijlFree( )`** the cleanup function.

Clean-up

After an application has finished using the IJL, the memory and other system resources allocated by the IJL should be released by calling the **`ijlFree( )`** function.

In the case of multiple encoding or decoding calls, as reviewed in the previous section, the **`ijlFree( )`** function should be called after the encode or decode function has been completed. This behavior will insure that resources will be properly cleaned up, and any values used by the IJL will not be corrupted.

Reading Data

ijlRead(JPEG_CORE_PROPERTIES *jcprops, IJLIOTYPE iotype) is one of two interface functions that access JPEG compressed data (the other is **ijlWrite( )** which is discussed in the following section).

The second parameter indicates the JPEG data location (i.e., a file or a buffer), the "mode of access", and any scaling to be applied during the decode process. The following two IJLIOTYPE naming conventions are used:

1. IJL_JBUFF_XXXX
   (Indicating the JPEG data is stored in a memory buffer).
2. IJL_JFILE_XXXX
   (Indicating the JPEG data is located in a standard I/O file).

When reading data, the mode of access must be READPARAMS, READHEADER, READENTROPY, READWHOLEIMAGE, READONEHALF, READONEQUARTER, READONEEIGHTH, or READTHUMBNAIL. Each of these is described in the tables below.

IJLIOTYPE	Description
IJL_JXXXX_READPARAMS	Indicates that JPEG parameters (i.e., height, width, number of channels, subsampling) are to be determined from the JPEG bit stream. For example, the following markers are parsed: SOI | [tables/misc] like APPn and DQT | SOF | [tables/misc] like DHT | stops at SOS Note: bit stream must start with SOI marker.

continued

IJLIOTYPE	Description
IJL_JXXXX_READHEADER	Indicates the Abbreviated Format for table specification data (i.e., Huffman tables, quantization tables, miscellaneous marker segments) is to be read. For example, the following markers are parsed: SOI | [tables/misc] | EOI (or stops at SOF or SOS) Note: bit stream must start with SOI marker.
IJL_JXXXX_READENTROPY	Indicates the Abbreviated Format for compressed image data is to be read. Identical to READWHOLEIMAGE except that the bit stream may or may not contain table specification data. For example, the following markers are parsed: SOI | [tables/misc] | SOF [tables/misc] like DHT | SOS | EOI Note: in this case only (READENTROPY), APP0 segments are skipped over.
IJL_JXXXX_READWHOLEIMAGE	Indicates the Interchange Format for compressed image data (i.e., the whole JPEG bit stream) is to be read. For example, the following markers are parsed: SOI | [tables/misc] like APPn and DQT | SOF [tables/misc] like DHT | SOS | EOI

Typically, `READPARAMS` is used to determine the JPEG's height and width in order to allocate an output buffer or for viewing reasons. Then, to read the remaining image data, `READWHOLEIMAGE` or `READENTROPY` is used.

READHEADER is commonly called to parse the Abbreviated Format for table specification data. It is subsequently paired with READENTROPY to obtain the Abbreviated Format for compressed image data. The READHEADER/READENTROPY pair is an optimal solution to Abbreviated Format JPEG decoding (i.e., for FlashPix* compressed images).

The IJLIOTYPE may also be used to indicate a scaled read (see *Scaled Decoding* for more information). The IJLIOTYPE enums for a scaled read have the same behavior, and may be used in the same way, as a READWHOLEIMAGE or READENTROPY. The following scaled decoding IJLIOTYPE's are defined:

IJLIOTYPE	Description
IJL_JXXXX_READONEHALF	Decodes the image scaled to ½ size. For example, the following markers are parsed: (See READENTROPY).
IJL_JXXXX_READONEQUARTER	Decodes the image scaled to ¼ size. For example, the following markers are parsed: (See READENTROPY).
IJL_JXXXX_READONEEIGHTH	Decodes the image scaled to 1/8 size. For example, the following markers are parsed: (See READENTROPY).

Lastly, the IJLIOTYPE may be used to indicate an attempt to decode an embedded thumbnail (if present) in a JFIF bit stream (see *Embedded Thumbnail Decoding* for more information). IJL_JXXXX_READTHUMBNAIL may be used in the same way as IJL_JXXXX_READPARAMS.

IJLIOTYPE	Description
IJL_JXXXX_READTHUMBNAIL	Attempts to decode an embedded thumbnail (if present) in a JFIF bit stream. For example, the following markers are parsed: (See READPARAMS).

When decoding a JPEG bit stream the following markers, and their corresponding segments if applicable, are not processed by the IJL (i.e., they are skipped over): APPn (except APP0 and APP14), DAC, DHP, DNL, EXP, JPGn, RES, SOFn (except SOF0, SOF1, and SOF2), and TEM. Any SOFn markers (except SOF0, SOF1, and SOF2) will cause the IJL_UNSUPPORTED_FRAME error.

Writing Data

ijlWrite(JPEG_CORE_PROPERTIES *, ILIOTYPE iotype) is the interface for writing data to a JPEG bit stream.

Similar to **ijlRead( )**, the second parameter indicates the JPEG data location (i.e., a file or a buffer) and the "mode of access". However, unlike **ijlRead( )**, the IJLIOTYPE parameter cannot be used to indicate scaled writing or to author embedded JFIF thumbnails. The following two IJLIOTYPE naming conventions are used:

1. IJL_JBUFF_XXXX
   (Indicating the JPEG compressed data is stored in a memory buffer).
2. IJL_JFILE_XXXX
   (Indicating the JPEG data is located in a standard I/O file).

When writing data, the mode of access must be `WRITEHEADER`, `WRITEENTROPY`, or `WRITEWHOLEIMAGE`. Each is described in the following table:

IJLIOTYPE	Description
IJL_JXXXX_WRITEHEADER	Indicates an Abbreviated Format for table specification data bit stream (i.e., Huffman tables, quantization tables, miscellaneous marker segments) is to be written. The following markers are authored: SOI | tables DQT and DHT | EOI
IJL_JXXXX_WRITEENTROPY	Indicates an Abbreviated Format for compressed image data bit stream is to be written. Identical to WRITEWHOLEIMAGE except that the bit stream may or may not contain table specification data. The following markers are authored: SOI | SOF | [DRI] | SOS | EOI
IJL_JXXXX_WRITEWHOLEIMAGE	Indicates a JPEG File Interchange Format (JFIF) for compressed image data bit stream is to be written (i.e., an entire JPEG using JFIF). The following markers are authored: SOI | tables/misc APP0, DQT, and DHT | SOF | [DRI] | SOS | EOI

`WRITEHEADER` is typically called to write a bit stream in the Abbreviated Format for table specification data. Also, it is usually paired with `WRITEENTROPY`, which is designed to write a bit stream in the Abbreviated Format for compressed image data. The `WRITEHEADER`/`WRITEENTROPY` pair is an optimal solution to Abbreviated Format JPEG encoding (i.e., for FlashPix compressed images).
When encoding data, the IJL writes the COM marker segment.
If the user comment is not specified, the default comment string "Intel® JPEG Library, [<version>]" will be written.

Opening a JPEG Image

Algorithm for “Normal Decoding of a JPEG Image”:

1. Allocate a `JPEG_CORE_PROPERTIES` data structure.
2. Initialize the IJL.
3. Get the JPEG image dimensions, etc.
4. Set up display parameters and allocate output storage.
5. Get the JPEG image data.
6. Close down the IJL.

In the following code segment, the IJL is used to decode a JPEG image from a JFIF file. Please refer to *Appendix B - Data Structure and Type Definitions* for additional details on IJL data structure definitions and default values, data type definitions, and error codes.

```
//-----------------------------------------------------------
// An example using the IntelR JPEG Library:
// -- Decode a JPEG image from a JFIF file to general pixel buffer.
//-----------------------------------------------------------

BOOL DecodeJPGFileToGeneralBuffer(
  LPCSTR lpszPathName,
  DWORD*  width,
  DWORD*  height,
  DWORD*  nchannels,
  BYTE**  buffer)
{
  BOOL   bres;
  IJLERR jerr;
  DWORD  x = 0; // pixels in scan line
  DWORD  y = 0; // number of scan lines
  DWORD  c = 0; // number of channels
  DWORD  wholeimagesize;
  BYTE*  pixel_buf = NULL;

  // Allocate the IJL JPEG_CORE_PROPERTIES structure.
  JPEG_CORE_PROPERTIES jcprops;

  bres = TRUE;
```

```
__try
{
  // Initialize the IntelR JPEG Library.
  jerr = ijlInit(&jcprops);
  if(IJL_OK != jerr)
  {
    bres = FALSE;
    __leave;
  }

  // Get information on the JPEG image
  // (i.e., width, height, and channels).
  jcprops.JPGFile = const_cast<LPSTR>(lpszPathName);

  jerr = ijlRead(&jcprops, IJL_JFILE_READPARAMS);
  if(IJL_OK != jerr)
  {
    bres = FALSE;
    __leave;
  }

  // Set up local data.
  x = jcprops.JPGWidth;
  y = jcprops.JPGHeight;
  c = 3; // Decode into a 3 channel pixel buffer.

  // Compute size of desired pixel buffer.
  wholeimagesize = (x * y * c);

  // Allocate memory to hold the decompressed image data.
  pixel_buf = new BYTE [wholeimagesize];
  if(NULL == pixel_buf)
  {
    bres = FALSE;
    __leave;
  }

  // Set up the info on the desired DIB properties.
  jcprops.DIBWidth    = x;
  jcprops.DIBHeight   = y; // Implies a bottom-up DIB.
  jcprops.DIBChannels = c;
  jcprops.DIBColor    = IJL_BGR;
```

```
jcprops.DIBPadBytes = 0;
jcprops.DIBBytes    = pixel_buf;

// Set the JPG color space ... this will always be
// somewhat of an educated guess at best because JPEG
// is "color blind" (i.e., nothing in the bit stream
// tells you what color space the data was encoded from).
// However, in this example we assume that we are
// reading JFIF files which means that 3 channel images
// are in the YCbCr color space and 1 channel images are
// in the Y color space.
switch(jcprops.JPGChannels)
{
  case 1:
  {
    jcprops.JPGColor = IJL_G;
    break;
  }

  case 3:
  {
    jcprops.JPGColor = IJL_YCBCR;
    break;
  }

  default:
  {
    // This catches everything else, but no
    // color twist will be performed by the IJL.
    jcprops.DIBColor = (IJL_COLOR)IJL_OTHER;
    jcprops.JPGColor = (IJL_COLOR)IJL_OTHER;
    break;
  }
}

// Now get the actual JPEG image data into the pixel buffer.
jerr = ijlRead(&jcprops, IJL_JFILE_READWHOLEIMAGE);
if(IJL_OK != jerr)
{
  bres = FALSE;
  __leave;
}
```

```
  } // __try
  __finally
  {
    if(FALSE == bres)
    {
      if(NULL != pixel_buf)
      {
        delete [] pixel_buf;
        pixel_buf = NULL;
      }
    }

    // Clean up the IntelR JPEG Library.
    ijlFree(&jcprops);

    *width     = x;
    *height    = y;
    *nchannels = c;
    *buffer    = pixel_buf;
  } // __finally

  return bres;
} // DecodeJPGFileToGeneralBuffer()
```

Note that the code segment above decodes a JPEG image into a "general pixel buffer" and thus no special allocation or alignment of the buffer is required. As previously mentioned in *Chapter 3, Architecture Description*, the IJL was designed to work with a general pixel buffer, and the user is responsible for the allocation of the buffer to hold the pixel data. The IJL in turn can write into, or read from, the buffer. The address of the buffer gets passed to the IJL through the **DIBBytes** field in the JPEG_CORE_PROPERTIES structure.

In the case that a user wants to decode into a Windows* DIB, the buffer size calculation above could possibly return an incorrect size. If a user wants to ensure the four (4) byte alignment of the buffer, as per the definition of a Windows DIB, he should use the IJL_DIB_PAD_BYTES macro included in the **ijl.h** header file. This macro definition is given by

```
#define IJL_DIB_PAD_BYTES(width,nchannels) \
```

```
  ( ( (width * nchannels) + (sizeof(DWORD) - 1) ) & (
~(sizeof(DWORD) - 1) ) - (width * nchannels) )
```

The corresponding `DIBPadBytes` value can be easily calculated as

```
jcprops.DIBPadBytes = IJL_DIB_PAD_BYTES(width,nchannels)
```

where width is the image width in pixels, and nchannels is the number of channels. The following code segment illustrates how to decode a JPEG image to a Windows DIB.

```
//-----------------------------------------------------------
// An example using the IntelR JPEG Library:
// -- Decode a JPEG image from a JFIF file to Windows DIB.
//-----------------------------------------------------------


BOOL DecodeJPGFileToDIB(
  LPCSTR            lpszPathName,
  BITMAPINFOHEADER** dib)
{
  BOOL   bres;
  IJLERR jerr;
  DWORD  width;
  DWORD  height;
  DWORD  nchannels;
  DWORD  dib_line_width;
  DWORD  dib_pad_bytes;
  DWORD  wholeimagesize;
  BYTE*             buffer = NULL;
  BITMAPINFOHEADER* bmih   = NULL;

  // Allocate the IJL JPEG_CORE_PROPERTIES structure.
  JPEG_CORE_PROPERTIES jcprops;

  bres = TRUE;

  __try
  {
    // Initialize the IntelR JPEG Library.
    jerr = ijlInit(&jcprops);
    if(IJL_OK != jerr)
```

```
{
  bres = FALSE;
  __leave;
}

// Get information on the JPEG image
// (i.e., width, height, and channels).
jcprops.JPGFile = const_cast<LPSTR>(lpszPathName);

jerr = ijlRead(&jcprops,IJL_JFILE_READPARAMS);
if(IJL_OK != jerr)
{
  bres = FALSE;
  __leave;
}

// Set up local data.
width     = jcprops.JPGWidth;
height    = jcprops.JPGHeight;
nchannels = 3; // Decode into a 3 channel pixel buffer.

// Compute DIB padding
dib_line_width = width * nchannels;
dib_pad_bytes  = IJL_DIB_PAD_BYTES(width,nchannels);

// Compute size of desired pixel buffer.
wholeimagesize = ( dib_line_width + dib_pad_bytes ) * height;

// Allocate memory to hold the decompressed image data.
buffer = new BYTE [sizeof(BITMAPINFOHEADER) + wholeimagesize];
if(NULL == buffer)
{
  bres = FALSE;
  __leave;
}

bmih = reinterpret_cast<BITMAPINFOHEADER*>(buffer);

bmih->biSize          = sizeof(BITMAPINFOHEADER);
bmih->biWidth         = width;
bmih->biHeight        = height;
bmih->biPlanes        = 1;
```

```
    bmih->biBitCount      = 24;
    bmih->biCompression   = BI_RGB;
    bmih->biSizeImage     = 0;
    bmih->biXPelsPerMeter = 0;
    bmih->biYPelsPerMeter = 0;
    bmih->biClrUsed       = 0;
    bmih->biClrImportant  = 0;

    // Set up the info on the desired DIB properties.
    jcprops.DIBWidth    = width;
    jcprops.DIBHeight   = height; // Implies a bottom-up DIB.
    jcprops.DIBChannels = nchannels;
    jcprops.DIBColor    = IJL_BGR;
    jcprops.DIBPadBytes = dib_pad_bytes;
    jcprops.DIBBytes    = reinterpret_cast<BYTE*>(buffer +
sizeof(BITMAPINFOHEADER));

    // Set the JPG color space ... this will always be
    // somewhat of an educated guess at best because JPEG
    // is "color blind" (i.e., nothing in the bit stream
    // tells you what color space the data was encoded from).
    // However, in this example we assume that we are
    // reading JFIF files which means that 3 channel images
    // are in the YCbCr color space and 1 channel images are
    // in the Y color space.
    switch(jcprops.JPGChannels)
    {
      case 1:
      {
        jcprops.JPGColor = IJL_G;
        break;
      }

      case 3:
      {
        jcprops.JPGColor = IJL_YCBCR;
        break;
      }

      default:
```

```
      {
        // This catches everything else, but no
        // color twist will be performed by the IJL.
        jcprops.DIBColor = (IJL_COLOR)IJL_OTHER;
        jcprops.JPGColor = (IJL_COLOR)IJL_OTHER;
        break;
      }
    }

    // Now get the actual JPEG image data into the pixel buffer.
    jerr = ijlRead(&jcprops,IJL_JFILE_READWHOLEIMAGE);
    if(IJL_OK != jerr)
    {
      bres = FALSE;
      __leave;
    }

  } // __try


  __finally
  {
    if(FALSE == bres)
    {
      if(NULL != buffer)
      {
        delete [] buffer;
        buffer = NULL;
      }
    }

    // Clean up the IntelR JPEG Library.
    ijlFree(&jcprops);

    *dib = bmih;
  } // __finally

  return bres;
} // DecodeJPGFileToDIB()
```

Creating a JPEG Image

Algorithm for “Normal Encoding of a JPEG Image”:

1. Initialize the IJL.
2. Set up encoding parameters (if different than the default values).
3. Write image data to the IJL.
4. Close the IJL.

The following code segment illustrates how to use the IJL to encode a JFIF image from a pixel buffer. Please refer to *Appendix B - Data Structure and Type Definitions* for additional details on IJL data structure definitions and default values, data type definitions, and error codes.

```
//-----------------------------------------------------------
// An example using the IntelR JPEG Library:
// -- Encode a JFIF file from Windows DIB.
//-----------------------------------------------------------


BOOL EncodeJPGFileFromDIB(
  LPCSTR           lpszPathName,
  BITMAPINFOHEADER* bmih)
{
  BOOL   bres;
  IJLERR jerr;
  DWORD  dib_pad_bytes;

  // Allocate the IJL JPEG_CORE_PROPERTIES structure.
  JPEG_CORE_PROPERTIES jcprops;

  bres = TRUE;

  __try
  {
    // Initialize the IntelR JPEG Library.
    jerr = ijlInit(&jcprops);
    if(IJL_OK != jerr)
```

```
    {
      bres = FALSE;
      __leave;
    }

    if(bmih->biBitCount != 24)
    {
      // not supported palette images
      bres = FALSE;
      __leave;
    }

    dib_pad_bytes  = IJL_DIB_PAD_BYTES(bmih->biWidth,3);

    // Set up information to write from the pixel buffer.
    jcprops.DIBWidth    = bmih->biWidth;
    jcprops.DIBHeight   = bmih->biHeight; // Implies a bottom-up DIB.
    jcprops.DIBBytes    = reinterpret_cast<BYTE*>(bmih) +
sizeof(BITMAPINFOHEADER);
    jcprops.DIBPadBytes = dib_pad_bytes;
    // Note: the following are default values and thus
    // do not need to be set.
    jcprops.DIBChannels = 3;
    jcprops.DIBColor    = IJL_BGR;


    jcprops.JPGFile = const_cast<LPSTR>(lpszPathName);

    // Specify JPEG file creation parameters.
    jcprops.JPGWidth  = bmih->biWidth;
    jcprops.JPGHeight = bmih->biHeight;

    // Note: the following are default values and thus
    // do not need to be set.

    jcprops.JPGChannels    = 3;
    jcprops.JPGColor       = IJL_YCBCR;
    jcprops.JPGSubsampling = IJL_411; // 4:1:1 subsampling.
    jcprops.jquality       = 75;      // Select "good" image quality

    // Write the actual JPEG image from the pixel buffer.
    jerr = ijlWrite(&jcprops,IJL_JFILE_WRITEWHOLEIMAGE);
```

```
    if(IJL_OK != jerr)
    {
      bres = FALSE;
      __leave;
    }

  } // __try


  __finally
  {
    // Clean up the IntelR JPEG Library.
    ijlFree(&jcprops);
  }

  return bres;
} // EncodeJPGFileFromDIB()
```

Interrupted Encoding and Decoding

The IJL is capable of interrupted encoding and decoding, and it may be interrupted at any time by asserting the "interrupt" flag in the `JPEG_PROPERTIES` data structure.

The IJL will return with status `IJL_INTERRUPT_OK` after completing processing on the current Minimum Coded Unit (MCU). The encoding or decoding process may be resumed at the same location by simply calling the appropriate **`ijlRead( )`** or **`ijlWrite( )`** function. The user may determine the location of the last decoded MCU via the **`left`** and **`top`** entries in the **`roi`** `IJL_RECT` structure inside of `JPEG_PROPERTIES`.

For example, the following code segment reads one MCU of JPEG data into a (previously specified) buffer, then it returns and repeats the process until the entire image has been decoded. This function can be used to periodically suspend the JPEG encoding or decoding process.

```
//-------------------------------------------------------------
// An example using the IntelR JPEG Library:
// -- Interrupted decoding.
//-------------------------------------------------------------
// In this example, we are doing full scale decoding.
// It could also be any of the scaled decoding modes.

BOOL DecodeJPGFileInterrupted(LPCSTR lpszPathName)
{
  BOOL   bres;
  IJLERR jerr;
  DWORD  width;
  DWORD  height;
  DWORD  nchannels;
  DWORD  wholeimagesize;
  BYTE*  pixel_buf = NULL;

  // Allocate the IJL JPEG_CORE_PROPERTIES structure.
  JPEG_CORE_PROPERTIES jcprops;

  bres = TRUE;

  __try
  {
    // Initialize the IntelR JPEG Library.
    jerr = ijlInit(&jcprops);
    if(IJL_OK != jerr)
    {
      bres = FALSE;
      __leave;
    }

    jcprops.JPGFile = const_cast<LPSTR>(lpszPathName);

    // Get information on the JPEG image
    // (i.e., width, height, and channels).
    jerr = ijlRead(&jcprops, IJL_JFILE_READPARAMS);
    if(IJL_OK != jerr)
    {
      bres = FALSE;
      __leave;
    }
```

```
// Set up local data.
width     = jcprops.JPGWidth;
height    = jcprops.JPGHeight;
nchannels = 3; // Decode into a 3 channel pixel buffer.

// Compute size of desired pixel buffer.
wholeimagesize = (width * height * nchannels);

// Allocate memory to hold the decompressed image data.
pixel_buf = new BYTE [wholeimagesize];
if(NULL == pixel_buf)
{
  bres = FALSE;
  __leave;
}

// Set up the info on the desired DIB properties.
jcprops.DIBWidth    = width;
jcprops.DIBHeight   = height; // Implies a bottom-up DIB.
jcprops.DIBChannels = nchannels;
jcprops.DIBColor    = IJL_BGR;
jcprops.DIBPadBytes = 0;
jcprops.DIBBytes    = pixel_buf;

// Set the JPG color space ... this will always be
// somewhat of an educated guess at best because JPEG
// is "color blind" (i.e., nothing in the bit stream
// tells you what color space the data was encoded from).
// However, in this example we assume that we are
// reading JFIF files which means that 3 channel images
// are in the YCbCr color space and 1 channel images are
// in the Y color space.
switch(jcprops.JPGChannels)
{
  case 1:
  {
    jcprops.JPGColor = IJL_G;
    break;
  }

  case 3:
```

```
    {
      jcprops.JPGColor = IJL_YCBCR;
      break;
    }

    default:
    {
      // This catches everything else, but no
      // color twist will be performed by the IJL.
      jcprops.DIBColor = (IJL_COLOR)IJL_OTHER;
      jcprops.JPGColor = (IJL_COLOR)IJL_OTHER;
      break;
    }
  }

  do
  {
    // Since the ROI values are updated following
    // an interrupt. We need to "reset" the ROI
    // values so that we continue to process over
    // the entire image.
    jcprops.jprops.roi.left   = 0;
    jcprops.jprops.roi.right  = 0;
    jcprops.jprops.roi.top    = 0;
    jcprops.jprops.roi.bottom = 0;

    jcprops.jprops.interrupt  = TRUE;

    jerr = ijlRead(&jcprops, IJL_JFILE_READENTROPY);

  } while(IJL_INTERRUPT_OK == jerr);

  //
  // ... now you probably want to do something with the
  // decompressed image like display it ...
  //

} // __try


__finally
{
```

```
    if(NULL != pixel_buf)
    {
      delete [] pixel_buf;
    }

    // Clean up the IntelR JPEG Library.
    ijlFree(&jcprops);
  }

  return bres;
} // DecodeJPGFileInterrupted()



//------------------------------------------------------------
// An example using the IntelR JPEG Library:
// -- Decode image row by row.
//------------------------------------------------------------
BOOL DecodeRowByRow(
  LPCSTR lpszJpgName,
  LPCSTR lpszBmpName)
{
  int                  cnt;
  int                  width;
  int                  height;
  int                  nchannels;
  int                  bmp_pad;
  int                  bmp_row_size;
  int                  bmp_buf_size;
  int                  current_row;
  BOOL                 bres;
  IJLERR               jerr;
  FILE*                out_file = NULL;
  BYTE*                bmp_bits = NULL;
  BYTE*                bmp_row = NULL;
  BYTE*                bmp_buf = NULL;
  LPBITMAPFILEHEADER   lpbmfh = NULL;
  LPBITMAPINFOHEADER   lpbmih = NULL;
  IJL_RECT             local_roi;
  JPEG_CORE_PROPERTIES jcprops;

  bres = TRUE;
```

```
  __try
  {
    // Initialize the Intel(R) JPEG Library.
    jerr = ijlInit(&jcprops);
    if(IJL_OK != jerr)
    {
      bres = FALSE;
      __leave;
    }

    jcprops.JPGFile = const_cast<LPSTR>(lpszJpgName);

    // Get information on the JPEG image (i.e., width, height, and
channels).
    jerr = ijlRead(&jcprops, IJL_JFILE_READPARAMS);
    if(IJL_OK != jerr)
    {
      bres = FALSE;
      __leave;
    }

    width     = jcprops.JPGWidth;
    height    = jcprops.JPGHeight;
    nchannels = 3;

    bmp_pad = IJL_DIB_PAD_BYTES(width,nchannels);

    bmp_row_size = (width * nchannels) + bmp_pad;

    // allocate buffer to hold one row DIB data
    bmp_row = new BYTE [bmp_row_size];

    if(NULL == bmp_row)
    {
      bres = FALSE;
      __leave;
    }

    memset(bmp_row,0,bmp_row_size);
```

```
    bmp_buf_size = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER)
+ bmp_row_size * height;

    // allocate buffer to hold entire DIB
    bmp_buf = new BYTE [bmp_buf_size];

    if(NULL == bmp_buf)
    {
      bres = FALSE;
      __leave;
    }

    bmp_bits = reinterpret_cast<BYTE*>(bmp_buf +
sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER));

    jcprops.DIBWidth    = width;
    jcprops.DIBHeight   = height;
    jcprops.DIBChannels = nchannels;
    jcprops.DIBColor    = IJL_BGR;
    jcprops.DIBPadBytes = bmp_pad;
    jcprops.DIBBytes    = bmp_row;

    // Set the JPG color space ... this will always be
    // somewhat of an educated guess at best because JPEG
    // is "color blind" (i.e., nothing in the bit stream
    // tells you what color space the data was encoded from).
    // However, in this example we assume that we are
    // reading JFIF files which means that 3 channel images
    // are in the YCbCr color space and 1 channel images are
    // in the Y color space.
    switch(jcprops.JPGChannels)
    {
      case 1:
      {
        jcprops.JPGColor = IJL_G;
        break;
      }

      case 3:
      {
        jcprops.JPGColor = IJL_YCBCR;
        break;
```

```
      }

      default:
      {
        // This catches everything else, but no
        // color twist will be performed by the IJL.
        jcprops.DIBColor = (IJL_COLOR)IJL_OTHER;
        jcprops.JPGColor = (IJL_COLOR)IJL_OTHER;
        break;
      }
    }


    //
    // Below is main code to decode image row by row
    //

    current_row = 0;

    do
    {
      // ROI is one row
      local_roi.left   = 0;
      local_roi.top    = current_row;
      local_roi.right  = width;
      local_roi.bottom = current_row + 1;

      jcprops.jprops.roi = local_roi;

      // decode ROI
      jerr = ijlRead(&jcprops, IJL_JFILE_READENTROPY);

      if(IJL_ROI_OK != jerr)
      {
        bres = FALSE;
        __leave;
      }

      // copy row data and reverse row order, to obtain bottom-left
DIB.
      memcpy(bmp_bits + (height - 1 - current_row) *
bmp_row_size,bmp_row,bmp_row_size);
```

```
      // advance to next row
      current_row++;

    } while(current_row != height);


    //
    // Now we have decoded image, and do anything on it.
    // For example write to file...
    //

    lpbmfh = reinterpret_cast<LPBITMAPFILEHEADER>(bmp_buf);

    lpbmfh->bfType      = 'MB';
    lpbmfh->bfSize      = bmp_buf_size;
    lpbmfh->bfReserved1 = 0;
    lpbmfh->bfReserved2 = 0;
    lpbmfh->bfOffBits   = sizeof(BITMAPFILEHEADER) +
sizeof(BITMAPINFOHEADER);

    lpbmih = reinterpret_cast<LPBITMAPINFOHEADER>(bmp_buf +
sizeof(BITMAPFILEHEADER));

    lpbmih->biSize          = sizeof(BITMAPINFOHEADER);
    lpbmih->biWidth         = width;
    lpbmih->biHeight        = height;
    lpbmih->biPlanes        = 1;
    lpbmih->biBitCount      = 24;
    lpbmih->biCompression   = BI_RGB;
    lpbmih->biSizeImage     = 0;
    lpbmih->biXPelsPerMeter = 0;
    lpbmih->biYPelsPerMeter = 0;
    lpbmih->biClrUsed       = 0;
    lpbmih->biClrImportant  = 0;

    out_file = fopen(lpszBmpName,"wb");

    if(NULL == out_file)
    {
      bres = FALSE;
      __leave;
```

```
    }

    cnt = fwrite(bmp_buf,sizeof(BYTE),lpbmfh->bfSize,out_file);

    if(cnt != lpbmfh->bfSize)
    {
      bres = FALSE;
      __leave;
    }

  } // __try

  __finally
  {
    if(NULL != bmp_row)
    {
      delete [] bmp_row;
    }

    if(NULL != bmp_buf)
    {
      delete [] bmp_buf;
    }

    if(NULL != out_file)
    {
      fclose(out_file);
    }

    // Clean up the IntelR JPEG Library.
    ijlFree(&jcprops);
  }

  return bres;
} // DecodeRowByRow()
```

```
//---------------------------------------------------------
// An example using the IntelR JPEG Library:
// -- Encode image by one MCU at a time.
//---------------------------------------------------------

/*
//  get_dib_parameters()
//
//  Purpose
//    gets image sizes from BMP file
//
//  Parameters
//    FILE* bmp_file  - input BMP file to gets data from
//    int*  width     - pointer to variable to store image width
//    int*  height    - pointer to variable to store image height
//    int*  nchannels - pointer to variable to store image number of
channels
//
//  Returns
//    0 - if read was successfully, if bmp_file is valid 24 bits per
pixel bitmap
//   -1 - if error has occured
//
*/

static int get_dib_parameters(
  FILE* bmp_file,
  int*  width,
  int*  height,
  int*  nchannels)
{
  int              res;
  int              cnt;
  BITMAPFILEHEADER bfh;
  BITMAPINFOHEADER bih;


  cnt = fread(&bfh,sizeof(BYTE),sizeof(BITMAPFILEHEADER),bmp_file);

  if(cnt != sizeof(BITMAPFILEHEADER))
  {
    res = -1;
```

```
    goto Exit;
  }

  if(bfh.bfType != 'MB')
  {
    res = -1;
    goto Exit;
  }

  cnt = fread(&bih,sizeof(BYTE),sizeof(BITMAPINFOHEADER),bmp_file);

  if(cnt != sizeof(BITMAPINFOHEADER))
  {
    res = -1;
    goto Exit;
  }

  if(bih.biBitCount != 24 || bih.biCompression != BI_RGB)
  {
    res = -1;
    goto Exit;
  }

  *width     = bih.biWidth;
  *height    = bih.biHeight;
  *nchannels = 3;

  res = 0;


Exit:

  return res;
} // get_dib_parameters()


/*
//  get_dib_chunk_data()
//
//  Purpose
//    gets chunk of data from BMP file.
//
```

```
//  Parameters
//    FILE* bmp_file       - input BMP file to gets data from
//    int   dib_chunk_size - size of chunk of data
//    BYTE* dib_chunk_ptr  - pointer to store data
//
//  Return
//    0 - if read was successfully, even if have reached the end of a
file
//   -1 - if error has occured
//
//  Note
//    It is assumed that the file pointer has a correct position.
//    For bottom-up DIBs, it is necessary to invert the order of scan
lines
//    that is read from a file. Here for simplification we do not make
it.
//
*/

static int get_dib_chunk_data(
  FILE* bmp_file,
  int   dib_chunk_size,
  BYTE* dib_chunk_ptr)
{
  int cnt;
  int res;

  res = 0;

  cnt = fread(dib_chunk_ptr,sizeof(BYTE),dib_chunk_size,bmp_file);

  if(cnt < dib_chunk_size)
  {
    res = ferror(bmp_file);
    if(0 != res)
    {
      res = -1;
    }
  }

  return res;
} // get_dib_chunk_data()
```

```
/*
//  ijl_compress_large_dib()
//
//  Purpose
//    to demonstrate one techniques to compress large DIBs
//    on mcu line by mcu line basis.
//
//  Parameters
//    char* bmp_file - ASCIIZ string with input BMP file name
//    char* jpg_file - ASCIIZ string with output JPG file name
//
//  Returns
//    0 - if success
//   -1 - if error has occured
//
*/

static int ijl_compress_large_dib(
  char* bmp_name,
  char* jpg_name)
{
  int                  i;
  int                  j;
  int                  res;
  int                  width;
  int                  height;
  int                  mcu_width;
  int                  mcu_height;
  int                  num_x_mcu;
  int                  num_y_mcu;
  int                  dib_line_size;
  int                  nchannels;
  int                  dib_chunk_size;
  BYTE*                dib_chunk_ptr;
  FILE*                bmp_file;
  IJLERR               jerr;
  JPEG_CORE_PROPERTIES jcprops;

  dib_chunk_ptr = NULL;

  bmp_file = fopen(bmp_name,"rb");
```

```
if(NULL == bmp_file)
{
  res = -1;
  goto Exit;
}

// read source image parameters
res = get_dib_parameters(bmp_file,&width,&height,&nchannels);

if(res != 0)
{
  goto Exit;
}

jerr = ijlInit(&jcprops);

if(IJL_OK != jerr)
{
  res = -1;
  goto Exit;
}

jcprops.DIBChannels    = nchannels;
jcprops.DIBWidth       = width;
jcprops.DIBHeight      = height;
jcprops.DIBPadBytes    = IJL_DIB_PAD_BYTES(width,nchannels);
jcprops.DIBColor       = IJL_BGR;
jcprops.DIBSubsampling = (IJL_DIBSUBSAMPLING)IJL_NONE;

jcprops.JPGFile        = jpg_name;

jcprops.JPGBytes       = NULL;
jcprops.JPGSizeBytes   = 0;

jcprops.JPGChannels    = nchannels;
jcprops.JPGWidth       = width;
jcprops.JPGHeight      = height;
jcprops.JPGColor       = IJL_YCBCR;
jcprops.JPGSubsampling = IJL_411;

jcprops.jquality       = 75;
```

```
  // sizes of mcu depend on subsampling
  switch(jcprops.JPGSubsampling)
  {
  case IJL_NONE:
    mcu_width  = 8;
    mcu_height = 8;
    break;

  case IJL_422:
    mcu_width  = 16;
    mcu_height = 8;
    break;

  case IJL_411:
    mcu_width  = 16;
    mcu_height = 16;
    break;

  default:
    res = -1;
    goto Exit;
  }

  // calculate number of mcu in image
  num_x_mcu = (width + mcu_width - 1) / mcu_width;
  num_y_mcu = (height + mcu_height - 1) / mcu_height;

  dib_line_size = width * nchannels +
IJL_DIB_PAD_BYTES(width,nchannels);

  dib_chunk_size = dib_line_size * mcu_height;

  // allocate memory to hold one mcu line
  dib_chunk_ptr = new BYTE [dib_chunk_size];

  if(NULL == dib_chunk_ptr)
  {
    res = -1;
    goto Exit;
  }

  // make illusion to IJL, that it is work with buffer
```

```
  jcprops.DIBBytes = dib_chunk_ptr;

  // process num_y_mcu line
  for(j = 0; j < num_y_mcu; j++)
  {
    // get next mcu line from BMP file
    res = get_dib_chunk_data(bmp_file,dib_chunk_size,dib_chunk_ptr);

    if(res != 0)
    {
      goto Exit;
    }

    // it is actually used pointer
    jcprops.jprops.state.DIB_ptr = dib_chunk_ptr;

    // process num_x_mcu in mcu line
    for(i = 0; i < num_x_mcu; i++)
    {
      // interrupt after each mcu
      jcprops.jprops.interrupt = 1;

      // compress current mcu (advance pointer to next mcu is internal
job)
      jerr = ijlWrite(&jcprops,IJL_JFILE_WRITEWHOLEIMAGE);

      if(IJL_INTERRUPT_OK == jerr)
      {
        // current mcu was encoded successfully
        continue;
      }

      if(IJL_OK == jerr)
      {
        // job is complete: all image is processed
        res = 0;
        break;
      }

      if(IJL_OK > jerr)
      {
        // error occured
```

```
        res = -1;
        break;
      }
    }
  }

  // if after processing num_y_mcu lines the library returns
IJL_INTERRUPT_OK,
  // it is to mean that some data are still keeping in internal
buffers. Need to flush it.
  if(IJL_INTERRUPT_OK == jerr)
  {
    // flush data from internal buffers
    jcprops.jprops.interrupt = 1;

    jerr = ijlWrite(&jcprops,IJL_JFILE_WRITEWHOLEIMAGE);

    if(IJL_OK != jerr)
    {
      res = -1;
      goto Exit;
    }
  }

  res = 0;

Exit:

  if(NULL != bmp_file)
  {
    fclose(bmp_file);
  }

  if(NULL != dib_chunk_ptr)
  {
    delete [] dib_chunk_ptr;
  }

  ijlFree(&jcprops);

  return res;
} // ijl_compress_large_dib()
```

Rectangle-of-Interest Decoding

Frequently only a portion of an image needs to be decompressed and displayed on the screen at any time. For example, a portion of a JPEG image may be displayed and "panned" at the user's request. Using this model, an application's architecture becomes much more efficient and the end-user gets to see the decoded image displayed in a significantly shorter amount of time.

To efficiently manage these situations, an application may request a rectangle-of-interest (ROI) to be decoded from the JPEG image by filling in the `IJL_RECT` structure in `JPEG_PROPERTIES` before decoding image data. Subsequent accesses to the IJL may be accelerated by simply modifying the ROI values and calling **`ijlRead( )`**.

The IJL uses several technologies designed to quickly access a given ROI in an image, and stores information from previous ROI passes to speed "panning" around an image.

The following code segment illustrates ROI decoding to fill an image buffer in two passes.

```
//-----------------------------------------------------------
// An example using the IntelR JPEG Library:
// -- Decode a JPEG image from a JFIF file using
// the Rectangle-Of-Interest (ROI) method.
//-----------------------------------------------------------

BOOL DecodeJPGFileByROI(LPCSTR lpszPathName)
{
  BOOL       bres;
  IJLERR     jerr;
  DWORD      width;
  DWORD      height;
  DWORD      nchannels;
  DWORD      wholeimagesize;
  BYTE*      pixel_buf = NULL;
  IJL_RECT   local_roi;

  // Allocate the IJL JPEG_CORE_PROPERTIES structure.
  JPEG_CORE_PROPERTIES jcprops;
```

```
bres = TRUE;

__try
{
  // Initialize the IntelR JPEG Library.
  jerr = ijlInit(&jcprops);
  if(IJL_OK != jerr)
  {
    bres = FALSE;
    __leave;
  }

  jcprops.JPGFile = const_cast<LPSTR>(lpszPathName);

  // Get information on the JPEG image
  // (i.e., width, height, and channels).
  jerr = ijlRead(&jcprops, IJL_JFILE_READPARAMS);
  if(IJL_OK != jerr)
  {
    bres = FALSE;
    __leave;
  }

  // Set up local data.
  width     = jcprops.JPGWidth;
  height    = jcprops.JPGHeight;
  nchannels = 3;

  // Decode into a 3 channel pixel buffer.
  // For this example, we will allocate an image buffer half
  // as big as the input image. Then, we will decode the
  // top half and bottom half of the image separately.
  // This could of course be extended to partition the image
  // into several rectangular tiles which would require a small
  // (or fixed size) image buffer. This technique yields
  // greatly increased memory performance for most
  // applications!
  wholeimagesize = width * ((height + 1) >> 1) * nchannels;

  // Allocate memory to hold the decompressed image data.
  pixel_buf = new BYTE [wholeimagesize];
```

```
if(NULL == pixel_buf)
{
  bres = FALSE;
  __leave;
}

// Set up the info on the desired DIB properties.
jcprops.DIBWidth    = width;
// Set a bottom-up DIB of half the original image size.
jcprops.DIBHeight   = (height + 1) >> 1;
jcprops.DIBChannels = nchannels;
jcprops.DIBColor    = IJL_BGR;
jcprops.DIBPadBytes = 0;
jcprops.DIBBytes    = pixel_buf;

// Set the JPG color space ... this will always be
// somewhat of an educated guess at best because JPEG
// is "color blind" (i.e., nothing in the bit stream
// tells you what color space the data was encoded from).
// However, in this example we assume that we are
// reading JFIF files which means that 3 channel images
// are in the YCbCr color space and 1 channel images are
// in the Y color space.
switch(jcprops.JPGChannels)
{
  case 1:
  {
    jcprops.JPGColor = IJL_G;
    break;
  }

  case 3:
  {
    jcprops.JPGColor = IJL_YCBCR;
    break;
  }

  default:
  {
    // This catches everything else, but no
    // color twist will be performed by the IJL.
    jcprops.DIBColor = (IJL_COLOR)IJL_OTHER;
```

```
    jcprops.JPGColor = (IJL_COLOR)IJL_OTHER;
    break;
  }
}

// Get the top half of the image.
local_roi.left   = 0;
local_roi.top    = 0;
local_roi.right  = width;
local_roi.bottom = (height + 1) >> 1;

jcprops.jprops.roi = local_roi;

// Now actually get the top half of the JPEG image data
// into the pixel buffer.
jerr = ijlRead(&jcprops, IJL_JFILE_READENTROPY);
if(IJL_ROI_OK != jerr)
{
  bres = FALSE;
  __leave;
}


// ... now you probably want to do something with the
// decompressed top half of the image like display it ...


// Next, get the bottom half of the image.
local_roi.left   = 0;
local_roi.top    = (height + 1) >> 1;
local_roi.right  = width;
local_roi.bottom = height;

jcprops.jprops.roi = local_roi;

// Now actually get the bottom half of the JPEG image data
// into the pixel buffer.
jerr = ijlRead(&jcprops, IJL_JFILE_READENTROPY);
if(IJL_ROI_OK != jerr)
{
  bres = FALSE;
  __leave;
```

```
    }

    // ... now you probably want to do something with the
    // decompressed bottom half of the image like display it ...
    //

  } // __try


  __finally
  {
    if(NULL != pixel_buf)
    {
      delete [] pixel_buf;
    }

    // Clean up the IntelR JPEG Library.
    ijlFree(&jcprops);
  }

  return bres;
} // DecodeJPGFileByROI()
```

Scaled Decoding

Most JPEG images can be efficiently decoded at 1/2, 1/4, or 1/8 the original image resolution. This is known as "scaled decoding", and it is typically at least two times faster than decoding an entire image. The IJL supports scaled decoding in parallel with rectangle-of-interest and interrupted decoding.

In practice, scaled decoding is very useful for generating "thumbnails" from JPEG images that do not already contain a thumbnail embedded in their bit stream.

The following table (Table 5-1) shows the calculations needed to determine the resulting scaled image size from an original JPEG image of size (Width x Height).

Table 5-1 Scaled Decoding Calculations

Scaled Decoding Type	Resulting Width' & Height'	I/O Type Specifier
1/2 Size	Width' = INT((Width + 1) / 2) Height' = INT((Height + 1) / 2)	IJL_JXXXX_READONEHALF
1/4 Size	Width' = INT((Width + 3) / 4) Height' = INT((Height + 3) / 4)	IJL_JXXXX_READONEQUARTER
1/8 Size	Width' = INT((Width + 7) / 8) Height' = INT((Height + 7) / 8)	IJL_JXXXX_READONEEIGHTH

To compute the size of the scaled image, use the following macro, included in the **ijl.h** file:

```
IJL_DIB_SCALE_SIZE(jpgsize, scale) =
    (((jpgsize)+(scale)-1)/(scale))
```

For example, an image of 2407 x 491 pixels would have a 1/8 scaled size of 301 x 62 pixels.

The following code illustrates scaled decoding of a JPEG image to generate a 1/8 sized version of the original JPEG image.

```
//-----------------------------------------------------------
// An example using the IntelR JPEG Library:
// -- Decode a JPEG image from a JFIF file.
// using the scaled decoding method.
//-----------------------------------------------------------

BOOL DecodeJPGFileOneEighth(LPCSTR lpszPathName)
{
  BOOL   bres;
  IJLERR jerr;
  DWORD  width;
  DWORD  height;
  DWORD  nchannels;
  DWORD  wholeimagesize;
  BYTE*  pixel_buf = NULL;

  // Allocate the IJL JPEG_CORE_PROPERTIES structure.
  JPEG_CORE_PROPERTIES jcprops;

  bres = TRUE;

  __try
  {
    // Initialize the IntelR JPEG Library.
    jerr = ijlInit(&jcprops);
    if(IJL_OK != jerr)
    {
      bres = FALSE;
      __leave;
    }

    jcprops.JPGFile = const_cast<LPSTR>(lpszPathName);

    // Get information on the JPEG image
    // (i.e., width, height, and channels).
    jerr = ijlRead(&jcprops, IJL_JFILE_READPARAMS);
    if(IJL_OK != jerr)
    {
      bres = FALSE;
      __leave;
```

```
}

// Set up local data.
// Note: In this case, width and height are rounded
// to the nearest factor of eight.
width     = (jcprops.JPGWidth  + 7) >> 3;
height    = (jcprops.JPGHeight + 7) >> 3;
nchannels = 3; // Decode into a 3 channel pixel buffer.

wholeimagesize = (width * height * nchannels);

// Allocate memory to hold the decompressed image data.
pixel_buf = new BYTE [wholeimagesize];
if(NULL == pixel_buf)
{
  bres = FALSE;
  __leave;
}

// Set up the info on the desired DIB properties.
jcprops.DIBWidth    = width;
jcprops.DIBHeight   = height; // Implies a bottom-up DIB.
jcprops.DIBChannels = nchannels;
jcprops.DIBColor    = IJL_BGR;
jcprops.DIBPadBytes = 0;
jcprops.DIBBytes    = pixel_buf;

// Set the JPG color space ... this will always be
// somewhat of an educated guess at best because JPEG
// is "color blind" (i.e., nothing in the bit stream
// tells you what color space the data was encoded from).
// However, in this example we assume that we are
// reading JFIF files which means that 3 channel images
// are in the YCbCr color space and 1 channel images are
// in the Y color space.
switch(jcprops.JPGChannels)
{
  case 1:
  {
    jcprops.JPGColor = IJL_G;
    break;
  }
```

```
    case 3:
    {
      jcprops.JPGColor = IJL_YCBCR;
      break;
    }

    default:
    {
      // This catches everything else, but no
      // color twist will be performed by the IJL.
      jcprops.DIBColor = (IJL_COLOR)IJL_OTHER;
      jcprops.JPGColor = (IJL_COLOR)IJL_OTHER;
      break;
    }
  }

  // Now get the actual JPEG image data into the pixel buffer
  // and scale the output to 1/8 th the original size.
  jerr = ijlRead(&jcprops, IJL_JFILE_READONEEIGHTH);
  if(IJL_OK != jerr)
  {
    bres = FALSE;
    __leave;
  }


  //
  // ... now you probably want to do something with the
  // decompressed scaled image like display it ...
  //

}

  __finally
{
  if(NULL != pixel_buf)
  {
    delete [] pixel_buf;
  }

  // Clean up the IntelR JPEG Library.
```

```
    ijlFree(&jcprops);
  }

  return bres;
} // DecodeJPGFileOneEighth()
```

Embedded Thumbnail Decoding

The IJL supports decoding some types of thumbnails embedded in JFIF compliant images. Specifically, the IJL supports decoding of uncompressed RGB thumbnails (either 1 byte/pixel or 3 bytes/pixel) as stored in accordance with the JFIF specification versions 1.01 and 1.02. Thumbnails compressed using JPEG are not supported at this time.

Before attempting to decode an embedded thumbnail, the user must provide a 24-bit DIB of at least 256x256 pixels. This is because the maximum dimensions of an embedded JFIF thumbnail are 256x256 pixels. Also, if the user wants the thumbnail decoded into packed 24-bit RGB values, then the **DIBColor** field needs to be set to IJL_RGB (otherwise it will be decoded into packed 24-bit BGR values).

Then, in order to actually decode the embedded thumbnail, the user needs to set the IJLIOTYPE parameter to IJL_JXXXX_READTHUMBNAIL when calling **ijlRead( )**. This IJLIOTYPE may be used interchangeably with IJL_JXXXX_READPARAMS on JFIF images. After this function call, the JPEG_CORE_PROPERTIES data structure is updated as follows:

1. The thumbnail's width and height (in pixels) are stored in the **JPGThumbWidth** and **JPGThumbHeight** fields (values of 0 indicate no embedded thumbnail present or an unsupported thumbnail), and
2. Decoded pixel values are placed into the buffer pointed to by the **DIBBytes** field.

In practice, embedded thumbnails have been only rarely found in standard (i.e., non-proprietary) formats in typical JPEG images. The IJL does not support proprietary embedded thumbnails.

Progressive Image Support

Decoding of Progressive DCT-based JPEG images is supported by the IJL. Progressive image decoding is transparent to the end user and requires no special support from the developer (i.e., the IJL does not support progressive display of the image).

Starting from version 1.5, the IJL supports authoring (encoding) of progressive images (note that restart intervals for encoding are not currently supported).
To create a progressive JPEG image, the user should call the library function **`ijlWrite()`** with the **`progressive_found`** field set to 1 in the `JPEG_PROPERTIES` structure. The following code sequence may serve as an example:

```
...
JPEG_CORE_PROPERTIES jcprops;
...
ijlInit(&jcprops);
....
   // Request to create a progressive image
jcprops.jprops.progressive_found = 1;
ijlWrite(&jcprops,IJL_JXXXX_WRITEWHOLEIMAGE);
```

The resulting image can be written either to a file (if **`ijlWrite()`** is called with second parameter set to `IJL_JFILE_WRITEWHOLEIMAGE`) or a previously allocated memory buffer (for calls with `IJL_JBUFF_WRITEWHOLEIMAGE`).

The progressive encoding algorithm, which can be either successive approximation or spectral selection, and the number of scans, are fixed in the library and cannot be changed by the user. These parameters are set depending on the number of channels and color space of the JPEG image. The library supports the following sets of progressive encoding parameters:

for 1-channel `IJL_G` images:
scan count is 6, with parameters per each pass as

```
1 scan; DC component 0; ss = 0, se = 63; ah = 0, al = 1
2 scan; AC component 0; ss = 1, se = 5;  ah = 0, al = 2
3 scan; AC component 0; ss = 6, se = 63; ah = 0, al = 2
4 scan; AC component 0; ss = 1, se = 63; ah = 2, al = 1
5 scan; DC component 0; ss = 0, se = 63; ah = 1, al = 0
6 scan; AC component 0; ss = 1, se = 63; ah = 1, al = 0
```

for 3-channel `IJL_YCBCR` images:
scan count is 10, with parameters per each pass as

```
1 scan; DC components 0,1,2; ss = 0, se = 63; ah = 0, al = 1
2 scan; AC component 0; ss = 1, se = 5; ah = 0, al = 2
3 scan; AC component 2; ss = 1, se = 63; ah = 0, al = 2
4 scan; AC component 1; ss = 1, se = 63; ah = 0, al = 2
5 scan; AC component 0; ss = 6, se = 63; ah = 0, al = 2
6 scan; AC component 0; ss = 1, se = 63; ah =2, al = 1
7 scan; DC components 0,1,2; ss = 0, se = 63; ah = 1, al = 0
8 scan; AC component 2; ss = 1, se = 63; ah = 1, al = 0
9 scan; AC component 1; ss = 1, se = 63; ah = 1, al = 0
10 scan; AC component 0; ss = 1, se = 63; ah = 1, al = 0
```

for 3-channel `IJL_RGB` images:
scan count is 8, with parameters per each pass as

```
1 scan; DC components 0,1,2; ss = 0, se = 63; ah = 0, al = 1
2 scan; AC component 0; ss = 1, se = 5; ah = 0, al = 0
3 scan; AC component 1; ss = 1, se = 5; ah = 0, al = 0
4 scan; AC component 2; ss = 1, se = 5; ah = 0, al = 0
5 scan; DC component 0,1,2; ss =0, se = 63; ah = 1, al = 0
6 scan; AC component 0; ss = 6, se = 63; ah =0, al = 0
7 scan; AC component 1; ss = 6, se = 63; ah = 0, al = 0
8 scan; AC component 2; ss = 6, se = 63; ah = 0, al = 0
```

for 3-channel `IJL_OTHER` images:
scan count is 8, with parameters per each pass as

```
1 scan; DC components 0,1,2; ss = 0, se = 63; ah = 0, al = 1
2 scan; AC component 0; ss = 1, se = 5; ah = 0, al = 0
3 scan; AC component 1; ss = 1, se = 5; ah = 0, al = 0
4 scan; AC component 2; ss = 1, se = 5; ah = 0, al = 0
5 scan; DC component 0,1,2; ss =0, se = 63; ah = 1, al = 0
6 scan; AC component 0; ss = 6, se = 63; ah =0, al = 0
7 scan; AC component 1; ss = 6, se = 63; ah = 0, al = 0
8 scan; AC component 2; ss = 6, se = 63; ah = 0, al = 0
```

for 4-channel IJL_RGBA_FPX images:
scan count is 10, with parameters per each pass as

```
1 scan; DC components 0,1,2,3; ss =0, se = 63; ah = 0, al = 1
2 scan; AC component 0; ss = 1, se = 5; ah = 0, al = 0
3 scan; AC component 1; ss = 1, se = 5; ah = 0, al = 0
4 scan; AC component 2; ss = 1, se = 5; ah = 0, al = 0
5 scan; AC component 3; ss = 1, se = 5; ah = 0, al = 0
6 scan; DC component 0,1,2,3; ss =0, se = 63; ah = 1, al = 0
7 scan; AC component 0; ss = 6, se = 63; ah =0, al = 0
8 scan; AC component 1; ss = 6, se = 63; ah = 0, al = 0
9 scan; AC component 2; ss = 6, se = 63; ah = 0, al = 0
10 scan; AC component 3; ss = 6, se = 63; ah = 0, al = 0
```

for 4-channel IJL_YCBCRA_FPX images:
scan count is 11, with parameters per each pass as

```
1 scan; DC components 0,1,2,3; ss =0, se = 63; ah = 0, al = 1
2 scan; AC component 0; ss = 1, se = 5; ah = 0, al = 2
3 scan; AC component 2; ss = 1, se = 63; ah = 0, al = 1
4 scan; AC component 1; ss = 1, se = 63; ah = 0, al = 1
5 scan; AC component 3; ss = 1, se = 63; ah = 0, al = 0
6 scan; AC component 0; ss = 6, se = 63; ah = 0, al = 2
7 scan; AC component 0; ss = 1, se = 63; ah =2, al = 1
8 scan; DC components 0,1,2,3; ss =0, se = 63; ah = 1, al = 0
9 scan; AC component 2; ss = 1, se = 63; ah = 1, al = 0
10 scan; AC component 1; ss = 1, se = 63; ah = 1, al = 0
11 scan; AC component 0; ss = 1, se = 63; ah = 1, al = 0
```

for 4-channel IJL_OTHER images:
scan count is 10, with parameters per each pass as

```
1 scan; DC components 0,1,2,3; ss =0, se = 63; ah = 0, al = 1
2 scan; AC component 0; ss = 1, se = 5; ah = 0, al = 0
3 scan; AC component 1; ss = 1, se = 5; ah = 0, al = 0
4 scan; AC component 2; ss = 1, se = 5; ah = 0, al = 0
5 scan; AC component 3; ss = 1, se = 5; ah = 0, al = 0
6 scan; DC component 0,1,2,3; ss =0, se = 63; ah = 1, al = 0
7 scan; AC component 0; ss = 6, se = 63; ah =0, al = 0
8 scan; AC component 1; ss = 6, se = 63; ah = 0, al = 0
9 scan; AC component 2; ss = 6, se = 63; ah = 0, al = 0
10 scan; AC component 3; ss = 6, se = 63; ah = 0, al = 0
```

In the above list we use the following notation:
`ss` – the first index in the spectral selection band;
`se` – the last index in the spectral selection band;
`ah` – the highest bit in the successive approximation;
`al` – the lowest bit in the successive approximation.

Accessing JPEG Images From a Buffer

JPEG is used as a compression standard in the FlashPix and TIFF 6.0 file formats, and the IJL supports decoding of data from these sources. FlashPix and/or TIFF codecs may extract JPEG data and provide a buffer (as opposed to a file) to the IJL, or they may require JPEG data to be buffered before output.
Note that the IJL allows JPEG data to be read from, or written to, a buffer in all access modes. Certain applications may find buffer-based JPEG access significantly faster than file-based JPEG access.

To write JPEG data to a buffer, do the following:

- Allocate a buffer of sufficient size (usually the buffer equal to the size of uncompressed data will suffice). If the buffer size is not enough, the IJL will return the error code `IJL_BUFFER_TOO_SMALL`.
- Set the necessary fields in the `JPEG_CORE_PROPERTIES` structure as
  `JPGFile` = `NULL`
  `JPGBytes` = pointer to the allocated buffer
  `JPGSizeBytes` = buffer size in bytes
- Call the **`ijlWrite()`** function with `IJL_JBUFF_WRITEWHOLEIMAGE` as its second parameter. On return, the buffer will contain the created JPEG data, and the `JPGSizeBytes` field will specify the actual JPEG data size in bytes. Note that earlier library versions returned incorrect JPEG data size in the buffer and replaced the pointer to the buffer. This bug was fixed in the current IJL version .

To decode JPEG data from a buffer, follow these steps:

- Get the buffer with JPEG data
- Set the fields in the `JPEG_CORE_PROPERTIES` structure as
  `JPGFile` = `NULL`

JPGBytes = pointer to the buffer with JPEG data
JPGSizeBytes = buffer size in bytes

- Call the **ijlRead()** function with IJL_JBUFF_READEWHOLEIMAGE as its second parameter.

The code examples below illustrate how to read JPEG data from a buffer, or write them to a buffer.

```
//-----------------------------------------------------------
// An example using the Intel(R) JPEG Library:
// -- Decode image from a JFIF buffer.
//-----------------------------------------------------------

BOOL DecodeFromJPEGBuffer(
  BYTE*   lpJpgBuffer,
  DWORD   dwJpgBufferSize,
  BYTE**  lppRgbBuffer,
  DWORD*  lpdwWidth,
  DWORD*  lpdwHeight,
  DWORD*  lpdwNumberOfChannels)
{
  BOOL   bres;
  IJLERR jerr;
  DWORD  dwWholeImageSize;
  BYTE*  lpTemp = NULL;

  // Allocate the IJL JPEG_CORE_PROPERTIES structure.
  JPEG_CORE_PROPERTIES jcprops;

  bres = TRUE;

  __try
  {
    // Initialize the Intel(R) JPEG Library.
    jerr = ijlInit(&jcprops);
    if(IJL_OK != jerr)
    {
      bres = FALSE;
      __leave;
    }

    // Get information on the JPEG image
    // (i.e., width, height, and channels).
```

```
jcprops.JPGFile      = NULL;
jcprops.JPGBytes     = lpJpgBuffer;
jcprops.JPGSizeBytes = dwJpgBufferSize;

jerr = ijlRead(&jcprops, IJL_JBUFF_READPARAMS);
if(IJL_OK != jerr)
{
  bres = FALSE;
  __leave;
}

// Set the JPG color space ... this will always be
// somewhat of an educated guess at best because JPEG
// is "color blind" (i.e., nothing in the bit stream
// tells you what color space the data was encoded from).
// However, in this example we assume that we are
// reading JFIF files which means that 3 channel images
// are in the YCbCr color space and 1 channel images are
// in the Y color space.

switch(jcprops.JPGChannels)
{
  case 1:
  {
    jcprops.JPGColor    = IJL_G;
    jcprops.DIBColor    = IJL_RGB;
    jcprops.DIBChannels = 3;
    break;
  }

  case 3:
  {
    jcprops.JPGColor    = IJL_YCBCR;
    jcprops.DIBColor    = IJL_RGB;
    jcprops.DIBChannels = 3;
    break;
  }

  default:
  {
    // This catches everything else, but no
    // color twist will be performed by the IJL.
```

```
        jcprops.JPGColor    = IJL_OTHER;
        jcprops.DIBColor    = IJL_OTHER;
        jcprops.DIBChannels = jcprops.JPGChannels;
        break;
      }
    }

    // Compute size of desired pixel buffer.
    dwWholeImageSize = jcprops.JPGWidth * jcprops.JPGHeight *
jcprops.DIBChannels;

    // Allocate memory to hold the decompressed image data.
    lpTemp = new BYTE [dwWholeImageSize];
    if(NULL == lpTemp)
    {
      bres = FALSE;
      __leave;
    }

    // Set up the info on the desired DIB properties.
    jcprops.DIBWidth    = jcprops.JPGWidth;
    jcprops.DIBHeight   = jcprops.JPGHeight;
    jcprops.DIBPadBytes = 0;
    jcprops.DIBBytes    = lpTemp;

    // Now get the actual JPEG image data into the pixel buffer.
    jerr = ijlRead(&jcprops, IJL_JBUFF_READWHOLEIMAGE);
    if(IJL_OK != jerr)
    {
      bres = FALSE;
      __leave;
    }

  } // __try

  __finally
  {
    if(FALSE == bres)
    {
      if(NULL != lpTemp)
      {
        delete [] lpTemp;
```

```
        lpTemp = NULL;
      }
    }

    // Clean up the Intel(R) JPEG Library.
    ijlFree(&jcprops);

    *lpdwWidth            = jcprops.DIBWidth;
    *lpdwHeight           = jcprops.DIBHeight;
    *lpdwNumberOfChannels = jcprops.DIBChannels;
    *lppRgbBuffer         = lpTemp;
  } // __finally

  return bres;
} // DecodeFromJPEGBuffer()


//------------------------------------------------------------
// An example using the Intel(R) JPEG Library:
// -- Encode Windows DIB to JPEG buffer.
//------------------------------------------------------------

BOOL EncodeToJPEGBuffer(
  BYTE*  lpRgbBuffer,
  DWORD  dwWidth,
  DWORD  dwHeight,
  BYTE** lppJpgBuffer,
  DWORD* lpdwJpgBufferSize)
{
  BOOL   bres;
  IJLERR jerr;
  DWORD  dwRgbBufferSize;
  BYTE*  lpTemp;

  // Allocate the IJL JPEG_CORE_PROPERTIES structure.
  JPEG_CORE_PROPERTIES jcprops;

  bres = TRUE;

  __try
  {
    // Initialize the Intel(R) JPEG Library.
```

```
jerr = ijlInit(&jcprops);
if(IJL_OK != jerr)
{
  bres = FALSE;
  __leave;
}

dwRgbBufferSize = dwWidth * dwHeight * 3;

lpTemp = new BYTE [dwRgbBufferSize];
if(NULL == lpTemp)
{
  bres = FALSE;
  __leave;
}

// Set up information to write from the pixel buffer.
jcprops.DIBWidth       = dwWidth;
jcprops.DIBHeight      = dwHeight; // Implies a bottom-up DIB.
jcprops.DIBBytes       = lpRgbBuffer;
jcprops.DIBPadBytes    = 0;
jcprops.DIBChannels    = 3;
jcprops.DIBColor       = IJL_RGB;

jcprops.JPGWidth       = dwWidth;
jcprops.JPGHeight      = dwHeight;
jcprops.JPGFile        = NULL;
jcprops.JPGBytes       = lpTemp;
jcprops.JPGSizeBytes   = dwRgbBufferSize;
jcprops.JPGChannels    = 3;
jcprops.JPGColor       = IJL_YCBCR;
jcprops.JPGSubsampling = IJL_411; // 4:1:1 subsampling.
jcprops.jquality       = 75; // Select "good" image quality

// Write the actual JPEG image from the pixel buffer.
jerr = ijlWrite(&jcprops,IJL_JBUFF_WRITEWHOLEIMAGE);
if(IJL_OK != jerr)
{
  bres = FALSE;
  __leave;
}
```

```
  } // __try

  __finally
  {
    if(FALSE == bres)
    {
      if(NULL != lpTemp)
      {
        delete[] lpTemp;
        lpTemp = NULL;
      }
    }

    *lppJpgBuffer      = lpTemp;
    *lpdwJpgBufferSize = jcprops.JPGSizeBytes;

    // Clean up the Intel(R) JPEG Library.
    ijlFree(&jcprops);

  }

  return bres;
} // EncodeToJPEGBuffer()
```

Working with Raw DCT Coefficients

Starting from version 1.5, the library can read from and write into a JPEG file the raw DCT coefficients. Here "raw DCT coefficients" mean the quantized DCT coefficients. You should specify the pointers to the external memory buffers for the DCT coefficients, using the `RAW_DATA_TYPES_STATE` structure which serves this purpose. It contains the array of pointers `raw_ptrs` and the control field `data_type`. To work with raw DCT coefficients, the `data_type` field must be set to 0.

The code examples below illustrate how to work with raw DCT coefficients: the first function decodes raw DCT coefficients from the JPEG file, and the second one creates a JPEG file from the raw DCT coefficients.

```
//---------------------------------------------------------
// An example using the Intel(R) JPEG Library:
// -- Decode raw DCT coefficients from the JPEG file;
//   create a JPEG file from the raw DCT coefficients.
//---------------------------------------------------------
#include "ijl.h"


#define MAX_RAW_QTBLS 4
#define MAX_RAW_HTBLS 8


typedef struct MY_PERSIST_STORAGE
{
  // raw quant tables
  JPEGQuantTable rawquanttables[MAX_RAW_QTBLS];
  unsigned char  quantizer[MAX_RAW_QTBLS][64];
  int            nqtables;
  int            maxquantindex;
  int            qprecision;

  // raw huffman tables
  JPEGHuffTable  rawhufftables[MAX_RAW_HTBLS];
  unsigned char  bits[MAX_RAW_HTBLS][16];
  unsigned char  vals[MAX_RAW_HTBLS][256];
  int            nhuffActables;
  int            nhuffDctables;
  int            maxhuffindex;

  // JPEG specific I/O data specifiers.
  int                JPGWidth;
  int                JPGHeight;
  int                JPGChannels;
  IJL_COLOR          JPGColor;
  IJL_JPGSUBSAMPLING JPGSubsampling;
  int                progressive_found;

  RAW_DATA_TYPES_STATE raw_coefs;

} MY_PERSIST_STORAGE;
```

```
void init_persist_storage(MY_PERSIST_STORAGE* ps)
{
  int i;

  // initialize raw quant tables
  for(i = 0; i < MAX_RAW_QTBLS; i++)
  {
    ps->rawquanttables[i].quantizer = &ps->quantizer[i][0];
    ps->rawquanttables[i].ident     = 0;
  }

  ps->nqtables      = 0;
  ps->maxquantindex = 0;
  ps->qprecision    = 0;

  // initialize raw huffman tables
  for(i = 0; i < MAX_RAW_HTBLS; i++)
  {
    ps->rawhufftables[i].bits       = &ps->bits[i][0];
    ps->rawhufftables[i].vals       = &ps->vals[i][0];
    ps->rawhufftables[i].ident      = 0;
    ps->rawhufftables[i].hclass     = 0;
  }

  ps->nhuffActables = 0;
  ps->nhuffDctables = 0;
  ps->maxhuffindex  = 0;

  ps->JPGWidth          = 0;
  ps->JPGHeight         = 0;
  ps->JPGChannels       = 0;
  ps->JPGColor          = IJL_OTHER;
  ps->JPGSubsampling    = IJL_NONE;
  ps->progressive_found = 0;

  ps->raw_coefs.data_type   = 0;
  ps->raw_coefs.raw_ptrs[0] = NULL;
  ps->raw_coefs.raw_ptrs[1] = NULL;
  ps->raw_coefs.raw_ptrs[2] = NULL;
  ps->raw_coefs.raw_ptrs[3] = NULL;
```

```
  return;
} // init_persist_storage()


void free_persist_storage(MY_PERSIST_STORAGE* ps)
{
  int i;

  // delete buffers for raw DCT coefs
  for(i = 0; i < 4; i++)
  {
    if(NULL != ps->raw_coefs.raw_ptrs[i])
    {
      delete [] ps->raw_coefs.raw_ptrs[i];
      ps->raw_coefs.raw_ptrs[i] = NULL;
    }
  }

  return;
} // free_persist_storage()


/*
//  get_raw_dct
//
//  Purpose
//    decode raw DCT coefficient from JPEG image
//
//  Parameters
//    input_file - input file name
//    ps         - pointer to persistent storage for raw DCT coefs
//
*/

int get_raw_dct(char* input_file,MY_PERSIST_STORAGE* ps)
{
  int  i;
  int  res;
  int  size;
  int  num_mcus;
  int  dct_block_size;
  IJLERR              jerr;
```

```
JPEG_CORE_PROPERTIES jcprops;

res = 0;

memset(&jcprops,0,sizeof(JPEG_CORE_PROPERTIES));

jerr = ijlInit(&jcprops);
if(IJL_OK != jerr)
{
  printf("ijlInit() failed - %s\n",ijlErrorStr(jerr));
  res = 1;
  goto Exit;
}

jcprops.JPGFile = input_file;

//
// supply buffers, to store raw quant tables
//

for(i = 0; i < MAX_RAW_QTBLS; i++)
{
  jcprops.jprops.rawquanttables[i].quantizer =
              ps->rawquanttables[i].quantizer;
}

//
// supply buffers, to store raw huffman tables
//

for(i = 0; i < MAX_RAW_HTBLS; i++)
{
  jcprops.jprops.rawhufftables[i].bits = ps->rawhufftables[i].bits;
  jcprops.jprops.rawhufftables[i].vals = ps->rawhufftables[i].vals;
}

//
// it will read raw JPEG tables to our buffers
//

jerr = ijlRead(&jcprops,IJL_JFILE_READPARAMS);
if(IJL_OK != jerr)
```

```
{
  printf("ijlRead(IJL_JFILE_READPARAMS) failed -
         %s\n",ijlErrorStr(jerr));
  res = 1;
  goto Exit;
}

//
// store info about raw quant tables
//

ps->nqtables      = jcprops.jprops.nqtables;
ps->maxquantindex = jcprops.jprops.maxquantindex;

//
// IJL can decode images with 16-bit quant tables,
// but can't encode with 16-bit quant tables.
// So, store this info, to check it before encoding
//

ps->qprecision    = jcprops.jprops.jFmtQuant[0].precision;

for(i = 0; i < jcprops.jprops.maxquantindex; i++)
{
  ps->rawquanttables[i].ident =
      jcprops.jprops.rawquanttables[i].ident;
}

//
// store info about raw huffman tables
//

ps->nhuffActables = jcprops.jprops.nhuffActables;
ps->nhuffDctables = jcprops.jprops.nhuffDctables;
ps->maxhuffindex  = jcprops.jprops.maxhuffindex;

for(i = 0; i < jcprops.jprops.maxhuffindex*2; i++)
{
  ps->rawhufftables[i].ident  =
      jcprops.jprops.rawhufftables[i].ident;
  ps->rawhufftables[i].hclass =
      jcprops.jprops.rawhufftables[i].hclass;
```

```
}

//
// store common JPEG image parameters to persist buffer
//

ps->JPGHeight      = jcprops.JPGHeight;
ps->JPGWidth       = jcprops.JPGWidth;
ps->JPGChannels    = jcprops.JPGChannels;
ps->JPGColor       = jcprops.JPGColor;
ps->JPGSubsampling = jcprops.JPGSubsampling;

//
// allocate memory for raw DCT coefs buffers
//

dct_block_size = sizeof(short) * 8 * 8;
num_mcus       = jcprops.jprops.numxMCUs * jcprops.jprops.numyMCUs;

for(i = 0; i < jcprops.JPGChannels; i++)
{
  int block_per_comp = jcprops.jprops.jframe.comps[i].hsampling *
                       jcprops.jprops.jframe.comps[i].vsampling;

  size = dct_block_size * block_per_comp * num_mcus;

  ps->raw_coefs.raw_ptrs[i] = new unsigned short [size];

  if(NULL == ps->raw_coefs.raw_ptrs[i])
  {
    printf("can't allocate memory for raw DCT coefs buffer\n");
    res = 1;
    goto Exit;
  }
}

//
// force read raw DCT coefs, instead full decoding
//

ps->raw_coefs.data_type  = 0;
jcprops.jprops.raw_coefs = &ps->raw_coefs;
```

```
  jerr = ijlRead(&jcprops,IJL_JFILE_READWHOLEIMAGE);
  if(IJL_OK != jerr)
  {
    printf("ijlRead(IJL_JFILE_READWHOLEIMAGE) failed -
           %s\n",ijlErrorStr(jerr));
    res = 0;
    goto Exit;
  }


  //
  // Note: for progressive images IJL can't store all huffman tables.
  //

  ps->progressive_found = jcprops.jprops.progressive_found;

Exit:

  jerr = ijlFree(&jcprops);
  if(IJL_OK != jerr)
  {
    printf("ijlFree() failed - %s\n",ijlErrorStr(jerr));
    res = 1;
  }

  return res;
} // get_raw_dct()


/*
//  put_raw_dct
//
//  Purpose
//    encode JPEG image from raw DCT coefficient
//
//  Parameters
//    output_file - output file name
//    ps          - pointer to persistent storage for raw DCT coefs
//
*/
```

```
int put_raw_dct(char* output_file,MY_PERSIST_STORAGE* ps)
{
  int  i;
  int  res = 0;
  IJLERR               jerr;
  JPEG_CORE_PROPERTIES jcprops;

  memset(&jcprops,0,sizeof(JPEG_CORE_PROPERTIES));

  jerr = ijlInit(&jcprops);
  if(IJL_OK != jerr)
  {
    printf("ijlInit() failed - %s\n",ijlErrorStr(jerr));
    res = 1;
    goto Exit;
  }

  jcprops.JPGFile = output_file;

  //
  // The IJL does not support encode with 16-bit quant tables,
  // so try to use default quant tables instead of 16-bit tables.
  //
  // Note, it can produce visible noise for image.
  //

  if(!ps->qprecision)
  {
    //
    // set custom raw quant tables
    //

    jcprops.jprops.nqtables      = ps->nqtables;
    jcprops.jprops.maxquantindex = ps->maxquantindex;

    for(i = 0; i < ps->nqtables; i++)
    {
      jcprops.jprops.rawquanttables[i].ident =
              ps->rawquanttables[i].ident;
      jcprops.jprops.rawquanttables[i].quantizer =
              ps->rawquanttables[i].quantizer;
    }
```

```
  jcprops.jprops.use_external_qtables = 1;
}

//
// Use default huffman tables for progressive images
//

if(!ps->progressive_found)
{
  //
  // set custom raw huffman tables
  //

  jcprops.jprops.nhuffActables = ps->nhuffActables;
  jcprops.jprops.nhuffDctables = ps->nhuffDctables;
  jcprops.jprops.maxhuffindex  = ps->maxhuffindex;

  for(i = 0; i < jcprops.jprops.maxhuffindex*2; i++)
  {
    jcprops.jprops.rawhufftables[i].ident =
                ps->rawhufftables[i].ident;
    jcprops.jprops.rawhufftables[i].hclass =
                ps->rawhufftables[i].hclass;
    jcprops.jprops.rawhufftables[i].bits =
                ps->rawhufftables[i].bits;
    jcprops.jprops.rawhufftables[i].vals =
                ps->rawhufftables[i].vals;
  }

  jcprops.jprops.use_external_htables = 1;
}
else
{
  //
  // let IJL use optimal huffman tables for progressive mode
  //

  jcprops.jprops.progressive_found = 1;
}

//
```

```
  // set common JPEG image parameters
  //

  jcprops.JPGHeight      = ps->JPGHeight;
  jcprops.JPGWidth       = ps->JPGWidth;
  jcprops.JPGChannels    = ps->JPGChannels;
  jcprops.JPGColor       = ps->JPGColor;
  jcprops.JPGSubsampling = ps->JPGSubsampling;

  if(jcprops.JPGChannels == 4)
  {
    jcprops.DIBColor    = IJL_RGBA_FPX;
    jcprops.DIBChannels = jcprops.JPGChannels;
  }

  //
  // write JPEG from raw DCT coefs, that are contained in raw_coefs
  //

  jcprops.jprops.raw_coefs = &ps->raw_coefs;

  //
  // jquality 50 - to not recalculate quant tables
  //

  jcprops.jquality = 50;

  jerr = ijlWrite(&jcprops,IJL_JFILE_WRITEWHOLEIMAGE);
  if(IJL_OK != jerr)
  {
    printf("ijlWrite(IJL_JFILE_WRITEWHOLEIMAGE) failed -
           %s\n",ijlErrorStr(jerr));
    res = 0;
    goto Exit;
  }

Exit:

  jerr = ijlFree(&jcprops);
  if(IJL_OK != jerr)
  {
    printf("ijlFree() failed - %s\n",ijlErrorStr(jerr));
```

```
    return 1;
  }

  return res;
} // put_raw_dct()
```

Support of a Pixel-Interleaved YCbCr422 Format

Starting from version 1.5, the library supports a pixel-interleaved YCbCr422 format both as an input for the encoder and as an output for the decoder. The decoder can decode into the pixel-interleaved 422 format from the 422 sampled JPEG file only. Resampling is not currently supported, which means that it is not possible to obtain data in the pixel-interleaved YCbCr422 format from the 444- or 411- sampled JPEG file.

```
//-----------------------------------------------------------
// An example using the Intel(R) JPEG Library:
// -- Decode image to YCbCr 422 format
//-----------------------------------------------------------
int ijl_decompress_to_ycbycr(
  char*  name,
  BYTE** buffer,
  int*   width,
  int*   height)
{
  int res;
  BYTE* dib_buffer = NULL;
  int   dib_buffer_size;
  IJLERR jerr;
  JPEG_CORE_PROPERTIES jcprops;

  jerr = ijlInit(&jcprops);
  if(IJL_OK != jerr)
  {
    printf("ijlInit() failed - %s\n",ijlErrorStr(jerr));
    res = 1;
    goto Exit;
  }

  jcprops.JPGFile        = name;
```

```
jerr = ijlRead(&jcprops,IJL_JFILE_READPARAMS);
if(IJL_OK != jerr)
{
  printf("ijlRead() failed - %s\n",ijlErrorStr(jerr));
  res = 1;
  goto Exit;
}

if(jcprops.JPGSubsampling != IJL_422 || jcprops.JPGChannels != 3)
{
  printf("only JPEG with 422 sampling can be decoded as YCBYCR DIB
          with 422 sampling\n");
  res = 1;
  goto Exit;
}

dib_buffer_size = jcprops.JPGWidth * jcprops.JPGHeight *
jcprops.JPGChannels;

dib_buffer = new BYTE [dib_buffer_size];
if(NULL == dib_buffer)
{
  printf("can't allocate memory\n");
  res = 1;
  goto Exit;
}

jcprops.DIBWidth       = jcprops.JPGWidth;
jcprops.DIBHeight      = -jcprops.JPGHeight;
jcprops.DIBChannels    = 2; // NOTE: we must set nchannels = 2
                            // to decode as YCBYCR
jcprops.DIBBytes       = dib_buffer;
jcprops.DIBPadBytes    = 0;
jcprops.DIBColor       = IJL_YCBCR;
jcprops.DIBSubsampling = IJL_422;

jerr = ijlRead(&jcprops,IJL_JFILE_READWHOLEIMAGE);
if(IJL_OK != jerr)
{
  printf("ijlRead() failed - %s\n",ijlErrorStr(jerr));
  res = 1;
  goto Exit;
```

```
  }

  res = 0;
  *buffer = dib_buffer;
  *width  = jcprops.JPGWidth;
  *height = jcprops.JPGHeight;

Exit:

  jerr = ijlFree(&jcprops);
  if(IJL_OK != jerr)
  {
    printf("ijlFree() failed - %s\n",ijlErrorStr(jerr));
    res = 1;
  }

  if(res == 1)
  {
    if(NULL != dib_buffer)
    {
      delete[] dib_buffer;
    }
  }

  return res;
} // ijl_decompress_to_ycbycr()


//-----------------------------------------------------------
// An example using the Intel(R) JPEG Library:
// -- Encode from YCbCr 422 format
//-----------------------------------------------------------
int ijl_compress_from_ycbycr(
  BYTE* buffer,
  int   width,
  int   height,
  char* name)
{
  int res = 0;
  IJLERR jerr;
  JPEG_CORE_PROPERTIES jcprops;
```

```
  jerr = ijlInit(&jcprops);
  if(IJL_OK != jerr)
  {
    printf("ijlInit() failed - %s\n",ijlErrorStr(jerr));
    res = 1;
    goto Exit;
  }

  jcprops.DIBWidth       = width;
  jcprops.DIBHeight      = -height;
  jcprops.DIBChannels    = 3;
  jcprops.DIBBytes       = buffer;
  jcprops.DIBPadBytes    = 0;
  jcprops.DIBColor       = IJL_YCBCR;
  jcprops.DIBSubsampling = IJL_422;

  jcprops.JPGFile        = name;
  jcprops.JPGWidth       = width;
  jcprops.JPGHeight      = height;
  jcprops.JPGChannels    = 3;
  jcprops.JPGColor       = IJL_YCBCR;
  jcprops.JPGSubsampling = IJL_422;
  jcprops.jquality       = 75;

  jerr = ijlWrite(&jcprops,IJL_JFILE_WRITEWHOLEIMAGE);
  if(IJL_OK != jerr)
  {
    printf("ijlInit() failed - %s\n",ijlErrorStr(jerr));
    res = 1;
    goto Exit;
  }

Exit:

  jerr = ijlFree(&jcprops);
  if(IJL_OK != jerr)
  {
    printf("ijlFree() failed - %s\n",ijlErrorStr(jerr));
    res = 1;
  }
  return res;
} // ijl_compress_from_ycbycr()
```

Odd Data Formats

Most of today's JPEG files are stored in the JPEG File Interchange Format (JFIF), and the IJL supports JFIF version 1.02. JFIF is a minimal file format that enables JPEG bit streams to be exchanged between a wide variety of platforms and applications. One feature of JFIF is that it specifies a standard color space. JFIF files are stored using either the 3-channel luminance/chrominance color space (YCbCr as defined by CCIR 601 (256 levels)), or the 1-channel grayscale color space (only the Y component of YCbCr).

However, the JPEG interchange format (not JFIF) defines compressed data storage formats that allow a great deal of flexibility to the representation of a set of data. A JPEG bit stream may have many meanings other than the common JFIF 3-channel, 2-D interleaved plane image data.

A JPEG image does not necessarily contain any information that specifies the color space of the image data. Any JPEG decoder is thus forced to make assumptions about the color format of some JPEG images. Modern file formats like TIFF 6.0 and FlashPix contain enough color space information to avoid this ambiguity.

- JPEG is often called "color blind". This is because nothing within a JPEG bit stream indicates what color format was used to encode the image data. When the color format of a JPEG image is unknown, or not supported by the IJL (i.e., Adobe's* CMYK), it is suggested that the user specify the `IJL_OTHER` color space format for both the **`JPGColor`** and **`DIBColor`** fields in the `JPEG_CORE_PROPERTIES` data structure. This technique prevents the IJL from applying a color space conversion. Then, it becomes the user's responsibility to perform their own color space manipulation (if so desired) outside of the IJL.
- If a JPEG bit stream indicates that data will be stored in separate planes, the IJL will present the data in a pixel-interleaved format. This may cause unexpected results, especially for data represented using multiple scans (i.e., one scan per block-row).

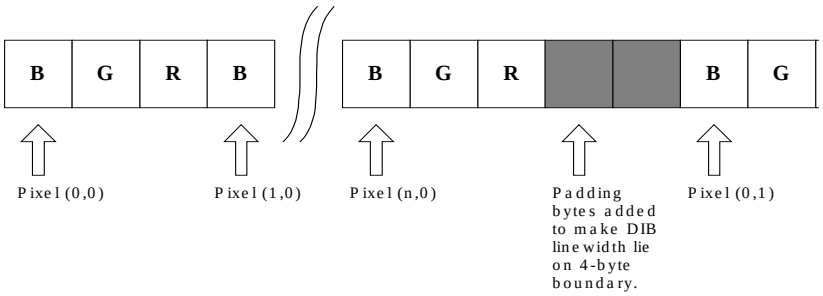
Pre- and Post-Processing

6

DIBs

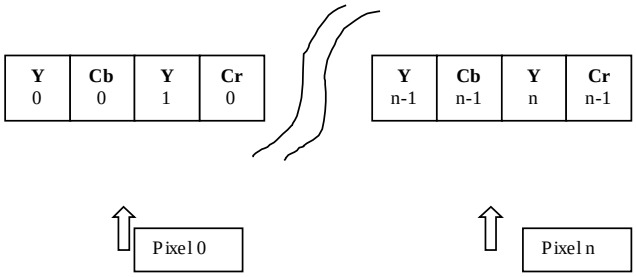
Image data in a Device Independent Bitmap (DIB) is stored in a byte interleaved form, one byte (8-bits) per channel. For the most common type, the Windows 24-bit DIB, the data is stored in a form graphically illustrated by Figure 6-1.

Figure 6-1 Windows 24-bit DIB Data Format



The IJL can receive as the encoder input and produce as the decoder output data in the 4:2:2 subsampled pixel interleaved format which is illustrated in Figure 6-2.

Figure 6-2 4:2:2 Subsampled Pixel-Interleaved Format



When authoring JPEG images, the IJL can receive input from a pixel buffer. Likewise, when decoding JPEG images, the IJL can send the output to a pixel buffer. The user has great freedom in specifying pixel buffer formats with regards to the number of color channels, the color space interpretation, and end-of-line padding.
The IJL supports input data with:

- Interleaved color planes.
- Non-subsampled data (with the exception of `IJL_YCBCR` color space, in which case only 4:2:2 subsampled pixel interleaved data are supported).
- Color channels from 1 to 255.
- Widths from 1 to 65,535.
- Heights from -65,535 to 65,535 (where values > 0 indicate a bottom-up DIB).
- End-of-line padding, or pad bytes, must be >= 0.

Additionally, for thumbnail output DIBs, the width and the absolute value of height must not exceed 255, and the color space must be either 3-channel `IJL_RGB` or `IJL_BGR`.

IJL Color Spaces

The following table (Table 6-1) illustrates the various DIB and JPEG color spaces supported by the IJL.

Table 6-1 IJL Supported Color Spaces

IJL Color Space	Valid IJL DIB Color Space?	Valid IJL JPEG Color Space?	Description
IJL_G	Yes	Yes	Grayscale (luminance only) 1 channel color space.
IJL_RGB	Yes	Yes	RGB (red-green-blue) 3 or 4 channel color space.
IJL_BGR	Yes	No	RGB 3 channel color space where the byte ordering has been reversed to BGR.
IJL_RGBA_FPX	Yes	Yes	FlashPix RGB 4 channel color space with pre-multiplied opacity.
IJL_YCBCR	Yes	Yes	CCIR 601 YCbCr (luminance-chrominance) 3 channel color space. Starting from version 1.5, the IJL supports the specific 4:2:2 subsampled pixel interleaved format used both as input data format for encoding, and output data format for decoding. In this case the data sequence is set to be Y0-Cb0-Y1-Cr0-Y2-Cb1-Y3-Cr1-... .
IJL_YCBCRA_FPX	No	Yes	FlashPix YCbCr 4 channel color space with pre-multiplied opacity and the YCbCr values are stored "flipped" (i.e., X' = 255 - X).
IJL_OTHER	Yes	Yes	Unknown color space where the user specifies the number of channels.

The `IJL_G` color space specifies that the DIB is stored in a Luminance only format with 8-bits per channel. The color space is defined as the Luminance (or Y) component of the standard YCbCr color space defined in CCIR 601 for 256 levels (8-bit) per channel.

The IJL_RGB color space follows the 8-bits per color channel definition of the RGB color space. Data is stored Red, Green, Blue from the lowest to the highest byte of a pixel.

The IJL_BGR color space is similar to the IJL_RGB color space except the byte order of the three channels are flipped. Data is stored Blue, Green, Red from the lowest to the highest bytes of a pixel. IJL_BGR is supported to provide fast input and output from standard Windows DIBs and Bitmaps (which use a BGR byte order).

The IJL_RGBA_FPX and IJL_YCBCRA_FPX color spaces are FlashPix 4 channel color spaces with pre-multiplied opacity and have been provided for greater compatibility with FlashPix JPEG compressed files.

The IJL_YCBCR color space is the standard YCbCr color space defined in CCIR 601 for 256 levels (8-bit) per channel. This is the color space used in most JPEG images and is supported by JFIF, EXIF, TIFF, FlashPix, and SPIFF file formats among others. It is strongly recommended that users author JPEG images in this color format (even when starting from a monochrome or grayscale source). The IJL_YCBCRA_FPX color space is not supported as valid DIB format for encoding.

The IJL_OTHER color space is used for user-defined or unknown DIB color spaces. The IJL will not perform any color space conversion when decoding JPEG images to an IJL_OTHER DIB color space. It will simply copy the appropriate number of channels from the source JPEG image.

Subsampling

The one (1) channel grayscale color space is not allowed to be subsampled.

Three (3) channel color spaces are allowed to be subsampled in either the 4:1:1 or the 4:2:2 formats. The 4:1:1 format is achieved by using a horizontal sampling factor of 2 and a vertical sampling factor of 2 in both the second and third channels. The 4:2:2 format is achieved by using a horizontal sampling factor of 2 and a vertical sampling factor of 1 in both the second and third channels. The non-subsampled format, or 1:1:1, is

denoted by a horizontal sampling factor of 1 and a vertical sampling factor of 1 in all three channels.

Four (4) channel color spaces are allowed to be subsampled in either the 4:1:1:4 or the 4:2:2:4 formats. The 4:1:1:4 format is achieved by using a horizontal sampling factor of 2 and a vertical sampling factor of 2 in both the second and third channels. The 4:2:2:4 format is achieved by using a horizontal sampling factor of 2 and a vertical sampling factor of 1 in both the second and third channels. The non-subsampled format, or 1:1:1:1, is denoted by a horizontal sampling factor of 1 and a vertical sampling factor of 1 in all four channels. The fourth channel, the alpha channel, is never allowed to be subsampled.

All neighboring pixels on a sampling interval are taken with equal weights to form the resulting value.

Upsampling

The IJL decompresses images that can have arbitrary sampling factors and maximum 10 blocks per each MCU, as compliant with JPEG standard. The default algorithm for decoding subsampled images is `IJL_BOX_FILTER`, which means that the decoded pixel value is simply replicated as many times as the sampling factors indicate. If both horizontal and vertical sampling factors do not exceed 2, you can use upsampling with triangular filter, which yields better results. For this purpose, set the `upsampling_type` field in the `JPEG_PROPERTIES` structure to `IJL_TRIANGLE_FILTER`.
In scaled decoding of subsampled images (see *Scaled Decoding*) with appropriately matching sampling factors (i.e. horizontal and vertical factors are equal and do not exceed 2), upsampling can be replaced by performing DCT of a larger size, which provides faster decoding with good image quality results. The IJL implements that approach, for instance, in case of scaled decoding of images at 1/2 size with upsampling 4:1:1.

Decoding and Post-Processing Matrix

The following table illustrates permitted color space decoding combinations and post-processing options in the IJL.

Table 6-2 IJL Decoding and Post-Processing Matrix

JPEG Color Space	JPEG Channels	DIB Color Space	DIB Channels	Format of Decoded Data	Post-Processing
IJL_G	1	IJL_G	1	Y, Y, …	CC No & US No
IJL_G	1	IJL_RGB	3	YYY, YYY, …	CC No & US No
IJL_G	1	IJL_BGR	3	YYY, YYY, … (see note 1 below)	CC No & US No
IJL_G	1	IJL_RGBA_FPX	4	YYYO, YYYO, …	CC No & US No
IJL_RGB	3	IJL_RGB	3	RGB, RGB, …	1:1:1 CC No & US No 4:1:1 CC No & US Yes 4:2:2 CC No & US Yes
IJL_RGB	3	IJL_BGR	3	BGR, BGR, …	1:1:1 CC No & US No 4:1:1 CC No & US Yes 4:2:2 CC No & US Yes

continued

Table 6-2 IJL Decoding and Post-Processing Matrix (continued)

JPEG Color Space	JPEG Channels	DIB Color Space	DIB Channels	Format of Decoded Data	Post-Processing
IJL_RGB	3	IJL_RGBA_FPX	4	RGBO, RGBO, …	1:1:1 CC No & US No 4:1:1 CC No & US Yes 4:2:2 CC No & US Yes
IJL_RGBA_FPX	4	IJL_RGBA_FPX	4	RGBA, RGBA, … (see note 2 below)	1:1:1:1 CC No & US No 4:1:1:4 CC No & US Yes 4:2:2:4 CC No & US Yes
IJL_YCBCR	3	IJL_G	1	Y, Y, …	1:1:1 CC No & US No 4:1:1 CC No & US Yes 4:2:2 CC No & US Yes
IJL_YCBCR	3	IJL_YCBCR	2	Y0-Cb0-Y1-Cr0-Y2-Cb1-... (see note 4 below)	4:2:2 CC No & US No
IJL_YCBCR	3	IJL_RGB	3	RGB, RGB, …	1:1:1 CC Yes & US No 4:1:1 CC Yes & US Yes 4:2:2 CC Yes & US Yes

continued

Table 6-2 IJL Decoding and Post-Processing Matrix (continued)

JPEG Color Space	JPEG Channels	DIB Color Space	DIB Channels	Format of Decoded Data	Post-Processing
IJL_YCBCR	3	IJL_BGR	3	BGR, BGR, …	1:1:1 CC Yes & US No 4:1:1 CC Yes & US Yes 4:2:2 CC Yes & US Yes
IJL_YCBCR	3	IJL_RGBA_FPX	4	RGBO, RGBO, …	1:1:1 CC Yes & US No 4:1:1 CC Yes & US Yes 4:2:2 CC Yes & US Yes
IJL_YCBCRA_FPX	4	IJL_RGBA_FPX	4	RGBA, RGBA, … (see note 3 below)	1:1:1:1 CC Yes & US No 4:1:1:4 C Yes & US Yes 4:2:2:4 C Yes & US Yes
IJL_OTHER	n	IJL_OTHER	1<=m<n	X0..X(m-1), X0..X(m-1), …	CC No & US if needed
IJL_OTHER	n	IJL_OTHER	m = n	X0..X(n-1), X0..X(n-1), …	CC No & US if needed
IJL_OTHER	n	IJL_OTHER	m > n	X0..X(n-1)En..E(m-1), X0..X(n-1)En..E(m-1), …	CC No & US if needed

Supporting Legend:

Symbol	Description
Y	Luminance channel
Cb	Cr chrominance channel (covering the red to blue-green range)
Cr	Cb chrominance channel (covering the blue to yellow range)
R	Red channel
G	Green channel
B	Blue channel
E	Empty value (i.e., the existing memory contents are not overwritten)
O	Opaque value (i.e., for 8-bit samples, it equals 255)
X	Any arbitrary channel value
CC	Color Space Conversion
US	Upsample
SS	Subsample

Supporting Notes:

1. Note, this is exactly the same as the `IJL_G` to `IJL_RGB` case.
2. Pursuant to the FlashPix specification, the pre-multiplied opacity is preserved.
3. Pursuant to the FlashPix specification, an "inverse flip" (that is, $X = 255 - X'$) is performed and the pre-multiplied opacity is preserved.
4. Starting from version 1.5, the IJL supports `IJL_YCBCR` DIB color space (currently for `DIBSubsampling` = `IJL_422` only). Decoding is implemented only for `JPGSubsampling` = `IJL_422.`

Encoding and Pre-Processing Matrix

The following table illustrates permitted color space encoding combinations and pre-processing options in the IJL.

Table 6-3 IJL Encoding and Pre-Processing Matrix

DIB Color Space	DIB Channels	JPEG Color Space	JPEG Channels	Format of Encoded Data	Pre-Processing
IJL_G	1	IJL_G	1	Y, Y, …	CC No & SS No
IJL_G	1	IJL_YCBCR	3	Y00, Y00, … (see note 1 below)	1:1:1 CC No & SS No 4:1:1 CC No & SS Yes 4:2:2 CC No & SS Yes
IJL_RGB	3	IJL_G	1	Y, Y, …	CC Yes & SS No
IJL_RGB	3	IJL_RGB	3	RGB, RGB, …	1:1:1 CC No & SS No 4:1:1 CC No & SS Yes 4:2:2 CC No & SS Yes

continued

Table 6-3 IJL Encoding and Pre-Processing Matrix (continued)

DIB Color Space	DIB Channels	JPEG Color Space	JPEG Channels	Format of Encoded Data	Pre-Processing
IJL_RGB	3	IJL_YCBCR	3	YCbCr, YCbCr, …	1:1:1 CC Yes & SS No 4:1:1 CC Yes & SS Yes 4:2:2 CC Yes & SS Yes
IJL_RGB	4	IJL_YCBCR	3	YCbCr, YCbCr, … (see note 2 below)	1:1:1 CC Yes & SS No 4:1:1 CC Yes & SS Yes 4:2:2 CC Yes & SS Yes
IJL_BGR	3	IJL_G	1	Y, Y, …	CC Yes & SS No
IJL_BGR	3	IJL_RGB	3	RGB, RGB, …	1:1:1 CC No & SS No 4:1:1 CC No & SS Yes 4:2:2 CC No & SS Yes
IJL_BGR	3	IJL_YCBCR	3	YCbCr, YCbCr, …	1:1:1 CC Yes & SS No 4:1:1 CC Yes & SS Yes 4:2:2 CC Yes & SS Yes
IJL_YCBCR	3	IJL_YCBCR	3	YCbCr, YCbCr, … (see note 5 below)	4:2:2 CC No & SS Yes

continued

Table 6-3 IJL Encoding and Pre-Processing Matrix (continued)

DIB Color Space	DIB Channels	JPEG Color Space	JPEG Channels	Format of Encoded Data	Pre-Processing
IJL_RGBA_FPX	4	IJL_RGBA_FPX	4	RGBA, RGBA, … (see note 3 below)	1:1:1:1 CC No & SS No 4:1:1:4 CC No & SS Yes 4:2:2:4 CC No & SS Yes
IJL_RGBA_FPX	4	IJL_YCBCRA_FPX	4	YCbCrA, YCbCrA, … (see note 4 below)	1:1:1:1 CC Yes & SS No 4:1:1:4 CC Yes & SS Yes 4:2:2:4 CC Yes & SS Yes
IJL_OTHER	n	IJL_OTHER	1<=m<n	X0..X(m-1), X0..X(m-1), …	CC No & SS if needed
IJL_OTHER	n	IJL_OTHER	m = n	X0..X(n-1), X0..X(n-1), …	CC No & SS if needed

Supporting Legend:

Symbol	Description
Y	Luminance channel
Cb	Cr chrominance channel (covering the red to blue-green range)
Cr	Cb chrominance channel (covering the blue to yellow range)
R	Red channel
G	Green channel
B	Blue channel
E	Empty value (i.e., the existing memory contents are not overwritten)
O	Opaque value (i.e., for 8-bit samples, it equals 255)
X	Any arbitrary channel value
CC	Color Space Conversion
US	Upsample
SS	Subsample

Supporting Notes:

1. The luminance values are retained and the chrominance values are set to zero.
2. Assumes no pre-multiplied opacity.
3. Pursuant to the FlashPix specification, the pre-multiplied opacity is preserved.
4. Pursuant to the FlashPix specification, a "flip" (i.e., X' = 255 - X) is performed and the pre-multiplied opacity is preserved.
5. The data encoding from `DIBColor` = `IJL_YCBCR` to `JPGColor` = `IJL_YCBCR` is currently supported only for `DIBSubsampling` = `IJL_422`.

Advanced IJL Features

7

This section describes some advanced features and imaging techniques that are possible with the IJL.

Use of Processor-Specific Code

The IJL detects the processor type and chooses the best available processor-specific code automatically (this is the default option). For example, if you use IJL on a system with Intel® Pentium® 4 processor, the library will take advantage of the code that has been specially optimized for that processor type.
However, you can direct the library to use the required code version by setting the USECPU key in the system registry to one of the following values:

0 - Blended code must be used (option for all legacy processors)
4 - Code optimized for Pentium II processor must be used
5 - Code optimized for Pentium III processor must be used
6 - Code optimized for Pentium 4 processor must be used

The USECPU key has the type DWORD and must be located at HKEY_LOCAL_MACHINE\Software\Intel Corporation\PLSuite\IJLib.

Setting the DCT Algorithm

The IJL supports two different DCT algorithms. The first one, set by `IJL_AAN` field value, is based on the work of Arai et al., see [Arai]. This algorithm is quite fast but has limited accuracy.

The second algorithm, which provides sufficient speed and higher accuracy, was derived from the Intel Integrated Performance Primitives for

Intel architecture. This is a default option, set after a call to **ijlInit()**. To use the previous version of the DCT algorithm, set the jcprops.jprops.dcttype field in JPEG_CORE_PROPERTIES structure to IJL_AAN. This setting must be done after calling **ijlInit()**, but prior to first call to **ijlRead()** or **ijlWrite()**.

Writing and Reading of JPEG Comment Block

Two new fields in the JPEG_CORE_PROPERTIES structure have been introduced: jpeg_comment is the pointer to a comment string, and jpeg_comment_size is the length in bytes of the comment string, including trailing zero. When IJL initialization takes place, these fields are set to 0. It means that the following predefined comment string will be inserted by the IJL while encoding data: "Intel® JPEG Library, [<version>]". If you need to insert your own comment for encoded data instead, set the pointer to the comment string and specify the length of the string. Similarly, to extract the comment from JPEG data while decoding, you should set the pointer to the comment buffer and specify the buffer size. If the comment string was successfully read and placed into the buffer, this field will be set to the number of bytes written into the buffer. In case the buffer has insufficient size, the IJL will write data until the buffer is full, and then return the error code IJL_ERR_COM_BUFFER.
If no comment string is present in JPEG data, the IJL will not change either buffer contents or the buffer size field.
The application program must both allocate and free memory for the comment string buffer.

Custom JPEG Tables

The IJL allows user-specified Huffman and quantization tables for specific authoring requirements. These tables are specified via entries in the JPEG_PROPERTIES data structure.

Custom Quantization Tables

The IJL can accept up to four custom quantization tables for authoring JPEG images. Quantization tables are specified in the IJL as an 8x8 array of 8-bit unsigned char entries in normal row-major, or non-zig-zagged, form. By default, the standard quantization tables are used in the IJL JPEG encoding procedures and are described as follows:

```
unsigned char DefaultLuminanceQuantTbl[] =
{
     16,  11,  12,  14,  12,  10,  16,  14,
     13,  14,  18,  17,  16,  19,  24,  40,
     26,  24,  22,  22,  24,  49,  35,  37,
     29,  40,  58,  51,  61,  60,  57,  51,
     56,  55,  64,  72,  92,  78,  64,  68,
     87,  69,  55,  56,  80, 109,  81,  87,
     95,  98, 103, 104, 103,  62,  77, 113,
    121, 112, 100, 120,  92, 101, 103,  99
};
unsigned char DefaultChrominanceQuantTbl[] =
{
    17,  18,  18,  24,  21,  24,  47,  26,
    26,  47,  99,  66,  56,  66,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99,
    99,  99,  99,  99,  99,  99,  99,  99
};
```

Each quantization factor is adjusted within IJL by a quality level multiplier and used to divide the input data to reduce its precision (and hence its storage size). The entries in the quantization arrays correspond to multipliers applied to certain spatial frequencies within the image. The lowest-order (DC) component is located in the upper-left hand corner.

The following code illustrates adding custom quantization tables prior to authoring a JPEG image.

```
//-------------------------------------------------------------
// An example using the IntelR JPEG Library:
// -- Author a JPEG image using custom quantization tables.
//-------------------------------------------------------------

// Your special quantization table goes here!
static BYTE HQLumQuantTable[] =
{
   16,  11,  12,  14,  12,  10,  16,  14,
   13,  14,  18,  17,  16,  19,  24,  40,
   26,  24,  22,  22,  24,  49,  35,  37,
   29,  40,  58,  51,  61,  60,  57,  51,
   56,  55,  64,  72,  92,  78,  64,  68,
   87,  69,  55,  56,  80, 109,  81,  87,
   95,  98, 103, 104, 103,  62,  77, 113,
  121, 112, 100, 120,  92, 101, 103,  99
};


// Your special quantization table goes here!
static BYTE HQChromQuantTable[] =
{
  17, 18, 18, 24, 21, 24, 47, 26,
  26, 47, 99, 66, 56, 66, 99, 99,
  99, 99, 99, 99, 99, 99, 99, 99,
  99, 99, 99, 99, 99, 99, 99, 99,
  99, 99, 99, 99, 99, 99, 99, 99,
  99, 99, 99, 99, 99, 99, 99, 99,
  99, 99, 99, 99, 99, 99, 99, 99,
  99, 99, 99, 99, 99, 99, 99, 99
};


BOOL EncodeJPGFileWithCustomQuantization(
  LPCSTR lpszPathName,
  DWORD   width,
  DWORD   height,
  DWORD   nchannels,
  BYTE*   pixel_buf)
{
  BOOL   bres;
  IJLERR jerr;
```

```
// Allocate the IJL JPEG_CORE_PROPERTIES structure.
JPEG_CORE_PROPERTIES jcprops;
  bres = TRUE;

__try
{
  // Initialize the IntelR JPEG Library.
  jerr = ijlInit(&jcprops);
  if(IJL_OK != jerr)
  {
    bres = FALSE;
    __leave;
  }

  // Set the custom quantization tables. For this example we
  // assign two custom tables, although up to four are possible.
  // Here we also assume the tables specify luminance and
  // chrominance quantization factors (as in a YCbCr image).
  jcprops.jprops.maxquantindex = 2;
  jcprops.jprops.nqtables = 2;
  jcprops.jprops.rawquanttables[ 0].quantizer = HQLumQuantTable;
  jcprops.jprops.rawquanttables[ 0].ident = 0;
  jcprops.jprops.rawquanttables[ 1].quantizer = HQChromQuantTable;
  jcprops.jprops.rawquanttables[ 1].ident = 1;
  jcprops.jprops.use_external_qtables = 1;

  // Now that we have assigned the tables, we need to decide which
  // color channels of the authored image will use which tables.
  // The ident member of rawquanttables specifies a unique
  // identifier for each table; we reference the quant_sel member of
  // each frame (image) component to this identifier.
  jcprops.jprops.jframe.comps[ 0].quant_sel = 0;
  jcprops.jprops.jframe.comps[ 1].quant_sel = 1;
  jcprops.jprops.jframe.comps[ 2].quant_sel = 1;
  jcprops.jprops.jframe.comps[ 3].quant_sel = 1;

  jcprops.DIBWidth    = width;
  jcprops.DIBHeight   = height;
  jcprops.DIBChannels = nchannels; // nchannels MUST BE 3!
  jcprops.DIBColor    = IJL_BGR;
  jcprops.DIBBytes    = pixel_buf;
```

```
    jcprops.JPGFile = const_cast<LPSTR>(lpszPathName);

    // Specify JPEG file creation parameters.
    jcprops.JPGWidth  = width;
    jcprops.JPGHeight = height;

    // Note: the following are default values and thus
    // do not need to be set.
    // jcprops.JPGChannels    = 3;
    // jcprops.JPGColor       = IJL_YCBCR;
    // jcprops.JPGSubsampling = IJL_411; // 4:1:1 subsampling.
    // jcprops.jquality       = 75;      // Select "good" image quality
    // Write the actual JPEG image from the pixel buffer.
    jerr = ijlWrite(&jcprops, IJL_JFILE_WRITEWHOLEIMAGE);
    if(IJL_OK != jerr)
    {
      bres = FALSE;
      __leave;
    }

  } // __try


  __finally
  {
    // Clean up the IntelR JPEG Library.
    ijlFree(&jcprops);
  }

  return bres;
} // EncodeJPGFileWithCustomQuantization()
```

The IJL formats the quantization tables for internal use before authoring any data. Thus, the tables that are passed to the IJL only need to persist as long as the first call to **`ijlWrite( )`**.

Saving the JPEG Quantization Tables

This feature has entered the IJL starting from version 1.5 of the library. Now JPEG quantization tables detected in the source data bit stream during decoding of a JPEG image can be saved into a user-defined buffer. To save quantization tables, you should specify pointers to desired storage buffers and then call the function **ijlRead()** in a usual way, as seen from the code example below:

```
JPEG_CORE_PROPERTIES jcprops;
..............
        BYTE quant_table1[64];
        BYTE quant_table2[64];
        BYTE quant_table3[64];
        BYTE quant_table4[64];

        jcprops.jprops.rawquanttables[0].quantizer = &quant_table1[0];
        jcprops.jprops.rawquanttables[1].quantizer = &quant_table2[0];
        jcprops.jprops.rawquanttables[3].quantizer = &quant_table3[0];
        jcprops.jprops.rawquanttables[4].quantizer = &quant_table4[0];
..............
        ijlRead( &jcprops, IJL_JXXX_READPARAMS);
```

Note that each quantization table has a fixed size of 64 bytes.
The non-progressive JPEG image can have up to four quantization tables. The IJL will accept all DQT segments and fill in the tables. The number of obtained quantization tables is stored in the jcprops.jprops.nqtables field. The IJL does not automatically free memory allocated for quantization tables storage buffers, so both allocation and deallocation should be done by your application program.

Custom Huffman Tables

The IJL accepts up to four sets of user-specified Huffman tables per authored image. Huffman tables are used to determine the entropy codes used in the run-length coding portion of the JPEG encoding process.

Huffman tables are specified in pairs: one table for each the DC and AC frequency components in an image channel. Each Huffman table requires two structures, one representing the bits required for each symbol, and one with the actual symbol values. The data format within each of these structures is identical to that of the embedded Huffman tables per the JPEG specification.

The following code illustrates image authoring using custom Huffman tables.

```
//-----------------------------------------------------------
// An example using the IntelR JPEG Library:
// -- Author a JPEG image using custom Huffman tables.
//-----------------------------------------------------------

// Your special Huffman DC Symbol Length table goes here!
static BYTE CustomLuminanceDCBits[] =
{
  0x00, 0x01, 0x05, 0x01, 0x01, 0x01, 0x01, 0x01,
  0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

// Your special Huffman DC Symbol table goes here!
static BYTE CustomLuminanceDCValues[] =
{
  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
0x0b
};

// Your special Huffman DC Symbol Length table goes here!
static BYTE CustomChrominanceDCBits[] =
{
  0x00, 0x03, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
  0x01, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00
```

```
};

// Your special Huffman DC Symbol table goes here!
static BYTE CustomChrominanceDCValues[] =
{
  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
0x0b
};

// Your special Huffman AC Symbol Length table goes here!
static BYTE CustomLuminanceACBits[] =
{
  0x00, 0x02, 0x01, 0x03, 0x03, 0x02, 0x04, 0x03,
  0x05, 0x05, 0x04, 0x04, 0x00, 0x00, 0x01, 0x7d
};

// Your special Huffman AC Symbol table goes here!
static BYTE CustomLuminanceACValues[] =
{
  0x01, 0x02, 0x03, 0x00, 0x04, 0x11, 0x05, 0x12,
  0x21, 0x31, 0x41, 0x06, 0x13, 0x51, 0x61, 0x07,
  0x22, 0x71, 0x14, 0x32, 0x81, 0x91, 0xa1, 0x08,
  0x23, 0x42, 0xb1, 0xc1, 0x15, 0x52, 0xd1, 0xf0,
  0x24, 0x33, 0x62, 0x72, 0x82, 0x09, 0x0a, 0x16,
  0x17, 0x18, 0x19, 0x1a, 0x25, 0x26, 0x27, 0x28,
  0x29, 0x2a, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39,
  0x3a, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49,
  0x4a, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59,
  0x5a, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69,
  0x6a, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79,
  0x7a, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89,
  0x8a, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98,
  0x99, 0x9a, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
  0xa8, 0xa9, 0xaa, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6,
  0xb7, 0xb8, 0xb9, 0xba, 0xc2, 0xc3, 0xc4, 0xc5,
  0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xd2, 0xd3, 0xd4,
  0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xe1, 0xe2,
  0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9, 0xea,
  0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
  0xf9, 0xfa
};
```

```
// Your special Huffman AC Symbol Length table goes here!
static unsigned char CustomChrominanceACBits[] =
{
  0x00, 0x02, 0x01, 0x02, 0x04, 0x04, 0x03, 0x04,
  0x07, 0x05, 0x04, 0x04, 0x00, 0x01, 0x02, 0x77
};

// Your special Huffman AC Symbol table goes here!
static unsigned char CustomChrominanceACValues[] =
{
  0x00, 0x01, 0x02, 0x03, 0x11, 0x04, 0x05, 0x21,
  0x31, 0x06, 0x12, 0x41, 0x51, 0x07, 0x61, 0x71,
  0x13, 0x22, 0x32, 0x81, 0x08, 0x14, 0x42, 0x91,
  0xa1, 0xb1, 0xc1, 0x09, 0x23, 0x33, 0x52, 0xf0,
  0x15, 0x62, 0x72, 0xd1, 0x0a, 0x16, 0x24, 0x34,
  0xe1, 0x25, 0xf1, 0x17, 0x18, 0x19, 0x1a, 0x26,
  0x27, 0x28, 0x29, 0x2a, 0x35, 0x36, 0x37, 0x38,
  0x39, 0x3a, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48,
  0x49, 0x4a, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58,
  0x59, 0x5a, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68,
  0x69, 0x6a, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78,
  0x79, 0x7a, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
  0x88, 0x89, 0x8a, 0x92, 0x93, 0x94, 0x95, 0x96,
  0x97, 0x98, 0x99, 0x9a, 0xa2, 0xa3, 0xa4, 0xa5,
  0xa6, 0xa7, 0xa8, 0xa9, 0xaa, 0xb2, 0xb3, 0xb4,
  0xb5, 0xb6, 0xb7, 0xb8, 0xb9, 0xba, 0xc2, 0xc3,
  0xc4, 0xc5, 0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xd2,
  0xd3, 0xd4, 0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda,
  0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9,
  0xea, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
  0xf9, 0xfa
};


BOOL EncodeJPGFileWithCustomHuffman(
  LPCSTR lpszPathName,
  DWORD   width,
  DWORD   height,
  DWORD   nchannels,
  BYTE*   pixel_buf)
{
  BOOL   bres;
```

```
IJLERR jerr;

// Allocate the IJL JPEG_CORE_PROPERTIES structure.
JPEG_CORE_PROPERTIES jcprops;

bres = TRUE;

__try
{
  // Initialize the IntelR JPEG Library.
  jerr = ijlInit(&jcprops);
  if(IJL_OK != jerr)
  {
    bres = FALSE;
    __leave;
  }

  // Set the custom Huffman tables. For this example, we
  // assign two sets of custom tables, though up to four are
  // possible. We also assume the tables specify luminance and
  // chrominance Huffman factors (as in a YCbCr image).
  jcprops.jprops.nhuffActables = 2;
  jcprops.jprops.nhuffDctables = 2;
  jcprops.jprops.maxhuffindex = 2;
  jcprops.jprops.rawhufftables[ 0].bits = CustomLuminanceDCBits;
  jcprops.jprops.rawhufftables[ 0].vals = CustomLuminanceDCValues;
  jcprops.jprops.rawhufftables[ 0].hclass = 0;
  jcprops.jprops.rawhufftables[ 0].ident = 0;
  jcprops.jprops.rawhufftables[ 1].bits = CustomLuminanceACBits;
  jcprops.jprops.rawhufftables[ 1].vals = CustomLuminanceACValues;
  jcprops.jprops.rawhufftables[ 1].hclass = 1;
  jcprops.jprops.rawhufftables[ 1].ident = 0;
  jcprops.jprops.rawhufftables[ 2].bits = CustomChrominanceDCBits;
  jcprops.jprops.rawhufftables[ 2].vals = CustomChrominanceDCValues;
  jcprops.jprops.rawhufftables[ 2].hclass = 0;
  jcprops.jprops.rawhufftables[ 2].ident = 1;
  jcprops.jprops.rawhufftables[ 3].bits = CustomChrominanceACBits;
  jcprops.jprops.rawhufftables[ 3].vals = CustomChrominanceACValues;
  jcprops.jprops.rawhufftables[ 3].hclass = 1;
  jcprops.jprops.rawhufftables[ 3].ident = 1;
  jcprops.jprops.use_external_htables = 1;
```

```
  // Now that we have assigned the tables, we need to decide which
  // channels of the authored image will use which tables.
  // The ident member of rawhufftables specifies a unique
  // identifier for each table; we reference the HuffIdentifier
  // member of each image (which applies to each component in
  // increasing order) to this identifier.
  jcprops.jprops.HuffIdentifierAC[ 0] = 0;
  jcprops.jprops.HuffIdentifierDC[ 0] = 0;
  jcprops.jprops.HuffIdentifierAC[ 1] = 1;
  jcprops.jprops.HuffIdentifierDC[ 1] = 1;
  jcprops.jprops.HuffIdentifierAC[ 2] = 1;
  jcprops.jprops.HuffIdentifierDC[ 2] = 1;
  jcprops.jprops.HuffIdentifierAC[ 3] = 1;
  jcprops.jprops.HuffIdentifierDC[ 3] = 1;

  jcprops.DIBWidth    = width;
  jcprops.DIBHeight   = height;
  jcprops.DIBChannels = nchannels; // only 3 is valid
  jcprops.DIBColor    = IJL_BGR;
  jcprops.DIBBytes    = pixel_buf;

  // Specify JPEG file creation parameters.
  jcprops.JPGWidth  = width;
  jcprops.JPGHeight = height;

  jcprops.JPGFile = const_cast<LPSTR>(lpszPathName);

  // Note: the following are default values and thus
  // do not need to be set.
  // jcprops.JPGChannels    = 3;
  // jcprops.JPGColor       = IJL_YCBCR;
  // jcprops.JPGSubsampling = IJL_411; // 4:1:1 subsampling.
  // jcprops.jquality       = 75;      // Select "good" image quality
  // Write the actual JPEG image from the pixel buffer.
  jerr = ijlWrite(&jcprops, IJL_JFILE_WRITEWHOLEIMAGE);
  if(IJL_OK != jerr)
  {
    bres = FALSE;
    __leave;
  }

} // __try
```

```
  __finally
  {
    // Clean up the IntelR JPEG Library.
    ijlFree(&jcprops);
  }

  return bres;
} // EncodeJPGFileWithCustomHuffman()
```

The IJL formats the Huffman tables for internal use before authoring any data. Thus, the tables passed to the IJL only need to persist as long as the first call to **ijlWrite( )**.

Saving the JPEG Huffman Tables

This feature has entered the IJL starting from version 1.5 of the library. Now JPEG Huffman tables detected in the source data bit stream during decoding of a JPEG image can be saved into a user-defined buffer. To save Huffman tables, you should specify pointers to desired storage buffers and then call the function **ijlRead()** in a usual way, as seen from the code example below:

```
JPEG_CORE_PROPERTIES jcprops;
..............
        BYTE huff_bits1[16];
        BYTE huff_vals1[256];
        BYTE huff_bits2[16];
        BYTE huff_vals2[256];
        BYTE huff_bits3[16];
        BYTE huff_vals3[256];
        BYTE huff_bits4[16];
        BYTE huff_vals4[256];
        BYTE huff_bits5[16];
        BYTE huff_vals5[256];
        BYTE huff_bits6[16];
        BYTE huff_vals6[256];
        BYTE huff_bits7[16];
```

```
        BYTE huff_vals7[256];
        BYTE huff_bits8[16];
        BYTE huff_vals8[256];

        jcprops.jprops.rawhufftables[0].bits = &huff_bits1[0];
        jcprops.jprops.rawhufftables[0].vals = &huff_vals1[0];
        jcprops.jprops.rawhufftables[1].bits = &huff_bits2[0];
        jcprops.jprops.rawhufftables[1].vals = &huff_vals2[0];
        jcprops.jprops.rawhufftables[2].bits = &huff_bits3[0];
        jcprops.jprops.rawhufftables[2].vals = &huff_vals3[0];
        jcprops.jprops.rawhufftables[3].bits = &huff_bits4[0];
        jcprops.jprops.rawhufftables[3].vals = &huff_vals4[0];
        jcprops.jprops.rawhufftables[4].bits = &huff_bits5[0];
        jcprops.jprops.rawhufftables[4].vals = &huff_vals5[0];
        jcprops.jprops.rawhufftables[5].bits = &huff_bits6[0];
        jcprops.jprops.rawhufftables[5].vals = &huff_vals6[0];
        jcprops.jprops.rawhufftables[6].bits = &huff_bits7[0];
        jcprops.jprops.rawhufftables[6].vals = &huff_vals7[0];
        jcprops.jprops.rawhufftables[7].bits = &huff_bits8[0];
        jcprops.jprops.rawhufftables[7].vals = &huff_vals8[0];
.............
        ijlRead( &jcprops, IJL_JXXX_READPARAMS);
```

Note that each Huffman table is defined by two data lists, `BITS` and `VALS`. The `BITS` list has a fixed size of 16 bytes, while the size of the respective `VALS` list is determined by the value contained in `BITS`. The `VALS` list can have the maximum size of 256 bytes.

The non-progressive JPEG image can have up to eight Huffman tables. The IJL will accept all DHT segments and fill in the tables. The number of obtained Huffman tables is stored in jcprops.jprops.nhuffDctables and jcprops.jprops.nhuffActables fields.

The IJL does not automatically free memory allocated for Huffman tables storage buffers, so both allocation and deallocation should be done by your application program.

Extended Baseline Decoding

This section describes techniques to persist formatted table information across multiple IJL accesses to minimize table processing and memory overhead.

Many image file formats separate the header, table, and entropy information of a JPEG stream. Some tile based formats, like FlashPix, may separate an image into tiles, each of which references JPEG tables stored elsewhere in the file. Optimal decoding requires that the table information is not processed for each tile in an image, rather the decoder formatted tables should be *persisted*. Persistence requires that after the Huffman and/or quantization tables are decoded and formatted, their formatted representation needs to be stored external to the IJL. Before an image is decoded, the formatted tables are then copied back into the appropriate locations within the `JPEG_PROPERTIES` structure.

For more information please refer to the white paper titled *"Using the IJL with JPEG Compressed FlashPix Files"*.

The `HUFFMAN_TABLE` and `QUANT_TABLE` structures contain Huffman and quantization tables in the proper decoder format. These tables are located within `JPEG_PROPERTIES` as specified in the following fragment:

```
////////////////////////////////////////////////////////////////
// ... a code fragment from the JPEG_PROPERTIES data structure ...
////////////////////////////////////////////////////////////////
     // Tables
     DWORD nqtables;
     DWORD maxquantindex;
     DWORD nhuffActables;
     DWORD nhuffDctables;
     DWORD maxhuffindex;
     QUANT_TABLE jFmtQuant[4];
     HUFFMAN_TABLE jFmtAcHuffman[4];
     HUFFMAN_TABLE jFmtDcHuffman[4];
     short* jEncFmtQuant[4];
     HUFFMAN_TABLE *jEncFmtAcHuffman[4];
     HUFFMAN_TABLE *jEncFmtDcHuffman[4];
```

```
// Allow user-defined tables.
DWORD use_default_htables;
DWORD use_default_qtables;
JPEGQuantTable rawquanttables[4];
JPEGHuffTable rawhufftables[8];
BYTE HuffIdentifierAC[4];
BYTE HuffIdentifierDC[4];
```

The important members for table persistence are **jFmtQuant**, **jFmtAcHuffman**, and **jFmtDcHuffman**. After decoding the tables using IJL_JXXXX_READHEADER, copy them to your persisted storage.

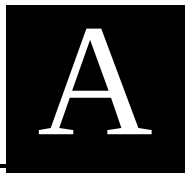
Next, to decode a JPEG bit stream (which is at a minimum assumed to be in the Abbreviated Format for compressed image data), the user copies the formatted tables back into the JPEG_PROPERTIES members and calls **ijlRead( )** with IJL_JXXXX_READWHOLEIMAGE.

Copying the persisted tables to JPEG_PROPERTIES is typically much faster than appending a table stream to the front of each JPEG data stream and forcing the decoder to process and format the tables at every call.

References

[Arai] Arai, Agui, and Nakajima, Trans. IEICE, vol. E 71(11), pp. 1095-1097, Nov. 1988.

Glossary of Terms



For purposes of this document, the following definitions apply.

Abbreviated Format (for compressed image data) – This format is identical to the Interchange Format, except that it may or may not include all tables required for decoding. This format is intended for use within applications where alternative mechanisms are available for supplying some or all of the table specification data needed for decoding.

Abbreviated Format (for table specification data) – This format contains only table specification data. It is a means by which the application may install in the decoder the tables required to subsequently reconstruct one or more images.

Baseline Mode – (a.k.a. *sequential DCT-based mode*) One of the four main categories of image compression processes defined by JPEG. This mode is the simplest DCT-based JPEG encoding and decoding process, and represents a minimum capability that must be present in all DCT-based JPEG decoders. Image components are compressed either individually or in groups in a single scan. Here is a summary of its essential characteristics:

- DCT-based process
- Source image: 8-bit samples within each component
- Sequential
- Huffman coding: 2 AC and 2 DC tables
- Decoders shall process scans with 1, 2, 3, and 4 components
- Interleaved and non-interleaved scans

Bit Stream - A partially encoded or decoded sequence of bits comprising an entropy-coded segment.

Channel – (a.k.a. *component*) A single color component of an image. An RGB image has 3 channels, a RGBA image has 4 channels, and a Grayscale image has only 1 channel.

Compressed Data – Either compressed image data or table specification data or both.

Compressed Image Data – A coded representation of an image as specified by the JPEG specification.

Continuous-tone Image – An image whose components have more than one bit per sample.

DCT – (Discrete Cosine Transform) A mathematical transformation using cosine basis functions which converts a block of samples into a corresponding array of basis function amplitudes.

DIB - (Device Independent Bitmap) A pixel buffer where the image data is stored in a byte interleaved form, one byte (8-bits) per channel. The most common type is the Windows 24-bit DIB.

Entropy Coding – A lossless procedure that converts a sequence of input symbols into a sequence of bits such that the average number of bits per symbol approaches the entropy of the input symbols.

Extended Baseline Mode - A sequential DCT-based encoding and decoding process in which additional capabilities are added beyond the Baseline mode. This mode extends the Baseline mode to a broader range of applications. Here is a summary of its essential characteristics:

DCT-based process
Source image: 8-bit or 12-bit samples
Sequential or progressive
Huffman or arithmetic coding: 4 AC and 4 DC tables
Decoders shall process scans with 1, 2, 3, and 4 components
Interleaved and non-interleaved scans

Grayscale Image – A continuous-tone image that has only one component.

Horizontal Sampling Factor – The relative number of horizontal data units of a particular component with respect to the number of horizontal data units in the other components.

Huffman Table – The set of variable length codes required in the Huffman coding process.

Huffman Coding – An entropy coding procedure that assigns a variable length code to each input symbol.

IJL - (Intel® JPEG Library) The IJL is a software library for application developers that provides high performance JPEG encoding and decoding of full color, and grayscale, still images. The IJL was developed to take advantage of MMX™ technology if present.

Interchange Format – (a.k.a. JPEG Interchange Format or JIF) A JPEG compressed image data bit stream that includes all tables that are required by the decoder (i.e., Huffman and quantization tables).

Interleaved – The descriptive term applied to the repetitive multiplexing of small groups of data units from each component in a scan in a specific order.

JFIF - (JPEG File Interchange Format) A minimal file format which enables JPEG bit streams to be exchanged between a wide variety of platforms and applications. The JFIF is entirely compatible with the standard JPEG Interchange Format.

JPEG - (Joint Photographic Experts Group) Usually refers to ISO DIS 10918-1 and 10918-2, “Digital compression and coding of continuous-tone still images", the compression standard this group created.

Lossless – A descriptive term for encoding and decoding processes and procedures in which the output of the decoding procedure(s) is identical to the input to the encoding procedure(s).

Lossy – A descriptive term for encoding and decoding processes which are not lossless.

MCU - (Minimum Coded Unit) The minimal set of data written to a compressed JPEG stream. The MCU is, for DCT-based JPEG coding processes, a set of rectangular regions over several channels representing the same pixel-based region. It is always a multiple of 8 pixels wide and high. Subsampling various color components of an image generates MCUs with dimensions greater than 8 x 8 pixels. For example, common 4:1:1 subsampled JPEG images have a 16 x 16 pixel MCU.

Non-Interleaved – The descriptive term applied to the data unit processing sequence when the scan has only one component.

Pixel Buffer - A rectangular array of pixels with each pixel having the same number of component values (color channels). The number of components and the color space interpretation of the components are also required.

Pre-Processing – The act of applying various operations to an image prior to sending it to the JPEG encoder. These operations typically include color space conversion and subsampling.

Post-Processing – The act of applying various operations to an image after receiving it from the JPEG decoder. These operations typically include upsampling and inverse color space conversion.

Progressive Mode – One of the four main categories of image compression processes defined by JPEG. This mode is a DCT-based coding process that is achieved by a sequence of scans, each of which codes part of the quantized DCT coefficient information.

Quantization - A lossy procedure in which the DCT coefficients are linearly scaled in order to achieve compression.

Quantization Table – The set of 64 integer values used to quantize the DCT coefficients.

Restart Interval – The integer number of MCUs processed as an independent sequence within a scan.

ROI - (Rectangle-of-Interest) A particular rectangular region of the image which can be specified by (top, left) and (bottom, right) pixel coordinates. The ROI must be contained within the image, but may encompass the total image.

Scan – A single pass through the data for one or more of the components in an image.

Subsampling – (a.k.a. Downsampling) A procedure by which the spatial resolution of an image is reduced.

Table Specification Data – The coded representation from which the tables used in the encoder and decoder are generated.

Upsampling – A procedure by which the spatial resolution of an image is increased.

Vertical Sampling Factor – The relative number of vertical data units of a particular component with respect to the number of vertical data units in the other components.

Zig-Zag Sequence – A specific sequential ordering of the DCT coefficients from (approximately) lowest spatial frequency to highest.

Data Structure and Type Definitions

B

For purposes of this document, the following definitions apply and are meant to be consistent with the IJL header file (**`ijl.h`**). If there are inconsistencies, the header file should always take precedence.

JPEG_CORE_PROPERTIES

```
/*D*
////////////////////////////////////////////////////////////////////////
//
// Name:  JPEG_CORE_PROPERTIES
//
// Purpose:  This is the primary data structure between the IJL and
// the external user.  It stores JPEG state information
// and controls the IJL.  It is user-modifiable.
//
// See the Developer's Guide for details on appropriate usage.
//
// Context:  Used by all low-level IJL routines to store
// pseudo-global information.
//
// Fields:
//  UseJPEGPROPERTIES  Set this flag != 0 if you wish to override
//                     the JPEG_CORE_PROPERTIES "IN" parameters with
//                     the JPEG_PROPERTIES parameters.
//
//  DIBBytes           IN:    Pointer to buffer of uncompressed data.
//  DIBWidth           IN:    Width of uncompressed data.
//  DIBHeight          IN:    Height of uncompressed data.
//  DIBPadBytes        IN:    Padding (in bytes) at end of each
//                            row in the uncompressed data.
//  DIBChannels        IN:    Number of components in the
//                            uncompressed data.
```

```
//  DIBColor            IN:     Color space of uncompressed data.
//  DIBSubsampling      IN:     Required to be IJL_NONE.
//
//  JPGFile             IN:     Pointer to file based JPEG.
//  JPGBytes            IN:     Pointer to buffer based JPEG.
//  JPGSizeBytes        IN:     Max buffer size. Used with JPGBytes.
//                      OUT:    Number of compressed bytes written.
//  JPGWidth            IN:     Width of JPEG image.
//                      OUT:    After reading (except READHEADER).
//  JPGHeight           IN:     Height of JPEG image.
//                      OUT:    After reading (except READHEADER).
//  JPGChannels         IN:     Number of components in JPEG image.
//                      OUT:    After reading (except READHEADER).
//  JPGColor            IN:     Color space of JPEG image.
//  JPGSubsampling      IN:     Subsampling of JPEG image.
//                      OUT:    After reading (except READHEADER).
//  JPGThumbWidth       OUT:    JFIF embedded thumbnail width [0-255].
//  JPGThumbHeight      OUT:    JFIF embedded thumbnail height [0-255].
//
//  cconversion_reqd    OUT:    If color conversion done on decode,
//                              TRUE.
//  upsampling_reqd     OUT:    If upsampling done on decode, TRUE.
//  jquality            IN:     [0-100] where highest quality is 100.
//
//  jprops              "Low-Level" IJL data structure.
//
////////////////////////////////////////////////////////////////////////
//
*D*/
struct JPEG_CORE_PROPERTIES
{
    DWORD UseJPEGPROPERTIES;              // default = 0

    // DIB specific I/O data specifiers.
    BYTE *DIBBytes;                       // default = NULL
    DWORD DIBWidth;                       // default = 0
    int DIBHeight;                        // default = 0
    DWORD DIBPadBytes;                    // default = 0
    DWORD DIBChannels;                    // default = 3
    IJL_COLOR DIBColor;                   // default = IJL_BGR
    IJL_DIBSUBSAMPLING DIBSubsampling;    // default = IJL_NONE
```

```
    // JPEG specific I/O data specifiers.
    char *JPGFile;                          // default = NULL
    BYTE *JPGBytes;                         // default = NULL
    DWORD JPGSizeBytes;                     // default = 0
    DWORD JPGWidth;                         // default = 0
    DWORD JPGHeight;                        // default = 0
    DWORD JPGChannels;                      // default = 3
    IJL_COLOR JPGColor;                     // default = IJL_YCBCR
    IJL_JPGSUBSAMPLING JPGSubsampling;      // default = IJL_411
    DWORD JPGThumbWidth;                    // default = 0
    DWORD JPGThumbHeight;                         // default = 0

    // JPEG conversion properties.
    DWORD cconversion_reqd;                 // default = TRUE
    DWORD upsampling_reqd;                  // default = TRUE
    DWORD jquality;                         // default = 75

    // Low-level properties.
    JPEG_PROPERTIES jprops;

};
```

Supporting Type Definitions

```
#define IJL_NONE                0
#define IJL_OTHER                   255

/*D*
////////////////////////////////////////////////////////////////////////
//
// Name:  IJLIOTYPE
//
// Purpose:  Possible types of data read/write/other operations to be
// performed by the functions ijlRead and ijlWrite.
//
// See the Developer's Guide for details on appropriate usage.
//
// Fields:
//
// IJL_JFILE_XXXXXXX  Indicates JPEG data in a stdio file.
//
// IJL_JBUFF_XXXXXXX  Indicates JPEG data in an addressable buffer.
//
////////////////////////////////////////////////////////////////////////
//
*D*/
typedef enum
{
     IJL_SETUP                  = -1,


     // Read JPEG parameters (i.e., height, width, channels,
     // sampling, etc.) from a JPEG bit stream.
     IJL_JFILE_READPARAMS       =  0,
     IJL_JBUFF_READPARAMS       =  1,

     // Read a JPEG Interchange Format image.
     IJL_JFILE_READWHOLEIMAGE   =  2,
     IJL_JBUFF_READWHOLEIMAGE   =  3,

     // Read JPEG tables from a JPEG Abbreviated Format bit stream.
     IJL_JFILE_READHEADER       =  4,
     IJL_JBUFF_READHEADER       =  5,
```

```
    // Read image info from a JPEG Abbreviated Format bit stream.
    IJL_JFILE_READENTROPY       =  6,
    IJL_JBUFF_READENTROPY       =  7,

    // Write an entire JFIF bit stream.
    IJL_JFILE_WRITEWHOLEIMAGE   =  8,
    IJL_JBUFF_WRITEWHOLEIMAGE   =  9,

    // Write a JPEG Abbreviated Format bit stream.
    IJL_JFILE_WRITEHEADER       = 10,
    IJL_JBUFF_WRITEHEADER       = 11,


    // Write image info to a JPEG Abbreviated Format bit stream.
    IJL_JFILE_WRITEENTROPY      = 12,
    IJL_JBUFF_WRITEENTROPY      = 13,


    // Scaled Decoding Options:

    // Reads a JPEG image scaled to 1/2 size.
    IJL_JFILE_READONEHALF       = 14,
    IJL_JBUFF_READONEHALF       = 15,

    // Reads a JPEG image scaled to 1/4 size.
    IJL_JFILE_READONEQUARTER    = 16,
    IJL_JBUFF_READONEQUARTER    = 17,

    // Reads a JPEG image scaled to 1/8 size.
    IJL_JFILE_READONEEIGHTH     = 18,
    IJL_JBUFF_READONEEIGHTH     = 19,

    // Reads an embedded thumbnail from a JFIF bit stream.
    IJL_JFILE_READTHUMBNAIL     = 20,
    IJL_JBUFF_READTHUMBNAIL     = 21

} IJLIOTYPE;


/*D*
////////////////////////////////////////////////////////////////////////
//
```

```
// Name:  IJL_COLOR
//
// Purpose:  Possible color space formats.
//
// Note these formats do *not* necessarily denote
// the number of channels in the color space.
// There exists separate "channel" fields in the
// JPEG_CORE_PROPERTIES data structure specifically
// for indicating the number of channels in the
// JPEG and/or DIB color spaces.
//
// See the Developer's Guide for details on appropriate usage.
//
////////////////////////////////////////////////////////////////////////
//
*D*/
typedef enum
{
     IJL_RGB = 1,        // Red-Green-Blue color space.
     IJL_BGR = 2,        // Reversed channel ordering from IJL_RGB.
     IJL_YCBCR = 3,      // Luminance-Chrominance color space as
                         // defined by CCIR Recommendation 601.
     IJL_G = 4,          // Grayscale color space.
     IJL_RGBA_FPX = 5,   // FlashPix RGB 4 channel color space that
                         // has pre-multiplied opacity.

     IJL_YCBCRA_FPX = 6,// FlashPix YCbCr 4 channel color space that
                         // has pre-multiplied opacity.

//   IJL_OTHER           // Some other color space not defined by
                         // the IJL.  This means no color space
                         // conversion will be done by the IJL.

} IJL_COLOR;


/*D*
////////////////////////////////////////////////////////////////////////
//
// Name:  IJL_JPGSUBSAMPLING
//
// Purpose:  Possible subsampling formats used in the JPEG.
```

```
//
// See the Developer's Guide for details on appropriate usage.
//
////////////////////////////////////////////////////////////////////////
//
*D*/
typedef enum
{
     IJL_411 = 1,        // Valid on a JPEG w/ 3 channels.
     IJL_422 = 2,        // Valid on a JPEG w/ 3 channels.

     IJL_4114 = 3,       // Valid on a JPEG w/ 4 channels.
     IJL_4224 = 4        // Valid on a JPEG w/ 4 channels.

//   IJL_NONE            // Corresponds to "No Subsampling".
                         // Valid on a JPEG w/ any number of channels.

//   IJL_OTHER           // Valid entry, but only used internally to
                         // the IJL.

} IJL_JPGSUBSAMPLING;


/*D*
////////////////////////////////////////////////////////////////////////
//
// Name:  IJL_DIBSUBSAMPLING
//
// Purpose:  Possible subsampling formats used in the DIB.
//
// See the Developer's Guide for details on appropriate usage.
//
////////////////////////////////////////////////////////////////////////
//
*D*/
typedef enum
{
//   IJL_NONE            = Corresponds to "No Subsampling".

} IJL_DIBSUBSAMPLING;
```

Return Error Codes

```
/*D*
////////////////////////////////////////////////////////////////////////
//
// Name:  IJLERR
//
// Purpose:  Listing of possible "error" codes returned by the IJL.
//
// See the Developer's Guide for details on appropriate usage.
//
// Context:  Used for error checking.
//
////////////////////////////////////////////////////////////////////////
//
*D*/
typedef enum
{
       // The following "error" values indicate an "OK" condition.
       IJL_OK                          =   0,
       IJL_INTERRUPT_OK                =   1,
       IJL_ROI_OK                          =   2,

       // The following "error" values indicate an error has occurred.
       IJL_EXCEPTION_DETECTED          =  -1,
       IJL_INVALID_ENCODER             =  -2,
       IJL_UNSUPPORTED_SUBSAMPLING     =  -3,
       IJL_UNSUPPORTED_BYTES_PER_PIXEL =  -4,
       IJL_MEMORY_ERROR                =  -5,
       IJL_BAD_HUFFMAN_TABLE           =  -6,
       IJL_BAD_QUANT_TABLE             =  -7,
       IJL_INVALID_JPEG_PROPERTIES     =  -8,
       IJL_ERR_FILECLOSE               =  -9,
       IJL_INVALID_FILENAME            = -10,
       IJL_ERROR_EOF                       = -11,
       IJL_PROG_NOT_SUPPORTED          = -12,
       IJL_ERR_NOT_JPEG                = -13,
       IJL_ERR_COMP                    = -14,
       IJL_ERR_SOF                     = -15,
       IJL_ERR_DNL                     = -16,
       IJL_ERR_NO_HUF                  = -17,
       IJL_ERR_NO_QUAN                 = -18,
```

```
        IJL_ERR_NO_FRAME                = -19,
        IJL_ERR_MULT_FRAME              = -20,
        IJL_ERR_DATA                    = -21,
        IJL_ERR_NO_IMAGE                = -22,
        IJL_FILE_ERROR                  = -23,
        IJL_INTERNAL_ERROR              = -24,
        IJL_BAD_RST_MARKER              = -25,
        IJL_THUMBNAIL_DIB_TOO_SMALL     = -26,
        IJL_THUMBNAIL_DIB_WRONG_COLOR   = -27,
        IJL_BUFFER_TOO_SMALL            = -28,
        IJL_UNSUPPORTED_FRAME           = -29,
        IJL_ERR_COM_BUFFER              = -30,
        IJL_RESERVED                    = -99

} IJLERR;
```

IJLibVersion Structure

```
/*D*
////////////////////////////////////////////////////
// Name:        IJLibVersion
//
// Purpose:     Stores library version info.
//
// Context:
//
// Example:
//   major           - 1
//   minor           - 0
//   build            - 1
//   Name            - "ijl10.dll"
//   Version         - "1.0.1 Beta1"
//   InternalVersion - "1.0.1.1"
//   BuildDate       - "Sep 22 1998"
//   CallConv        - "DLL"
//
////////////////////////////////////////////////////
*D*/

typedef struct _IJLibVersion
{
  int     major;
  int     minor;
  int     build;
  LPCSTR Name;
  LPCSTR Version;
  LPCSTR InternalVersion;
  LPCSTR BuildDate;
  LPCSTR CallConv;

} IJLibVersion;
```

Frequently Asked Questions

C

Q: I have a top-to-bottom image. Can IJL handle this type of DIBs?

A: Yes, the IJL supports both top-down and bottom-up image orientations for encoding. If an image file has bottom-up orientation, you need just to specify a negative value for the `DIBHeight` field in the `JPEG_CORE_PROPERTIES` structure. Note that JPEG data format defines only the top-down image orientation; thus, the `JPGHeight` field must always contain a positive value.

Q: Does IJL have a resize capability (I have a 600x400 DIB and I want to write a 300x200 JPEG image)?

A: The IJL supports scaled decoding mode to decode an image at 1/2, 1/4, or 1/8 of initial size. There is no provision in IJL for resizing an image while encoding. You can use Intel® Image Processing Library to resize a source image.

Q: I would like to use the DC and AC coefficients to check similarity of two JPEG images. Is it possible to retrieve the coefficients with the IJL?

A: Yes, starting from version 1.5, the library supports raw DCT coefficients retrieval. See "Working with Raw DCT Coefficients" section in Chapter 5 of this manual.

Q: Can you provide any information on a new version of your JPEG library that will support scanline based encoding? For our company's applications, having access to the entire bitmap for encoding is impractical. In some cases, our software deals with images that are hundreds of megabytes.

A: You can use interrupted encoding and decoding capability, which is supported by the IJL. See code examples in this manual, *Decoding an Image Row by Row*, and *Encoding by One MCU at a Time*.