Intel® JPEG Library

Coexistence with the Intel Image Processing Library

Version 1.1
July 9, 1999

Intel Corporation, 5200 N.E. Elam Young Parkway, Hillsboro, OR 97124-6497

Table of Contents

Table of Figures

1. Introduction

1.1 About This Document

This document demonstrates the interaction between the Intel JPEG Library and the Intel Performance Library's Image Processing Library (IPL). We illustrate the translation between IJL data structures and IPL data structures, and potential uses of the two libraries in a single application.

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 operating system, and the Microsoft Foundation Classes* application framework is also useful.

1.2 Nature of Product

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

The IJL was designed for use on Intel® architecture platforms and has been tuned for speed 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 that is not included in the ISO JPEG standard. This added functionality is typically necessary when working with JPEG images, and includes pre-processing and post-processing options like sampling and color space conversions.

1.3 Support and Feedback

You may submit questions and/or problems through our on-line customer support center on the IJL web site.
The URL is: http://developer.intel.com/software/products/perflib/ijl/ .

1.4 Minimum Requirements

- The IJL requires the presence of the Microsoft Windows* 95 or Microsoft Windows NT* operating system, and uses the Win32 application programming interface (API).
- 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 you use must be able to produce an application capable of calling functions contained in a Win32 DLL..
- The IJL was designed to run on at least a 90 MHz Intel® Pentium® processor.
- The IJL has been designed for high performance and efficient memory usage on IA platforms and will take full advantage of MMX technology if present.

2. Matching IJL and IPL data structures

The IPL uses data structures to represent the images that serve as the source and destination of image processing filters. IJL likewise uses members of its interface structure (JPEG_CORE_PROPERTIES) to specify input and output data formats. By carefully describing the inputs and outputs from both and addressing memory effectively we can closely merge IPL and IJL.

The following system diagrams illustrate common IJL-IPL usage models.

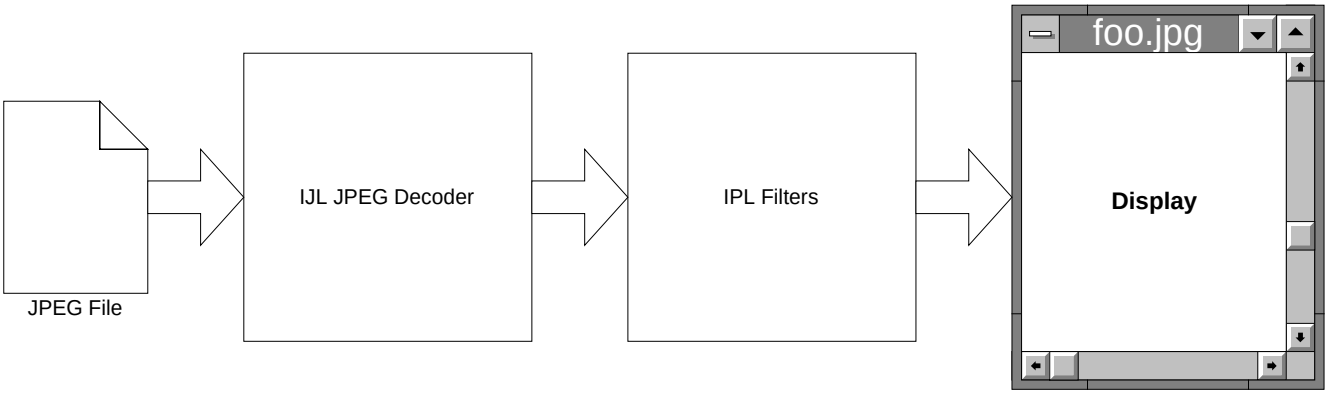

Figure 1. Using IJL to create an IPL Image from a JPEG file

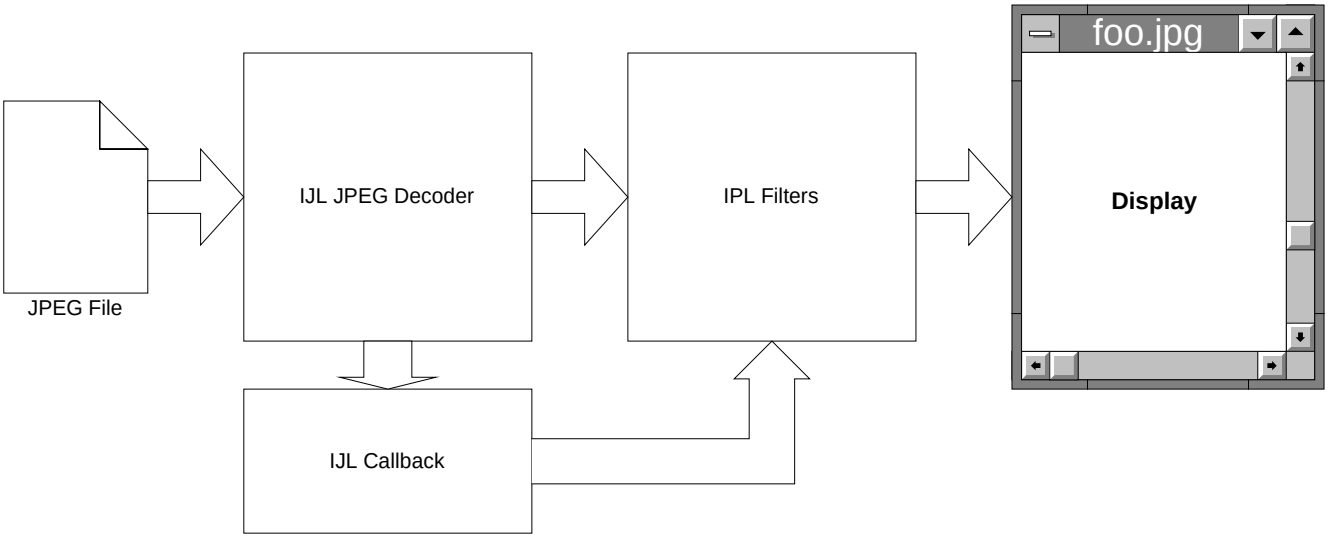

Figure 2. Using IJL to create a tile-based IPL Image from a JPEG file

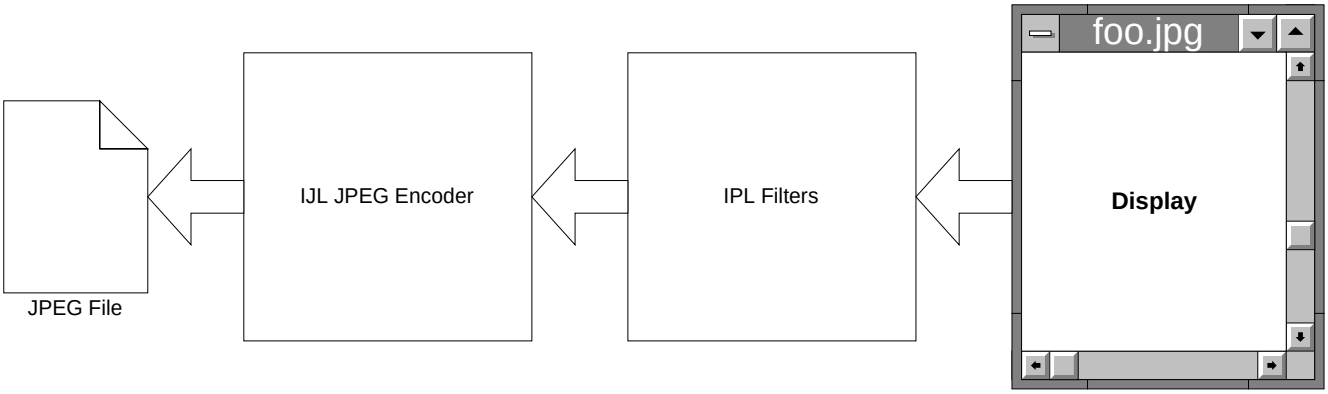

Figure 3. Using IJL to save an IPL Image to a JPEG file

Using IJL in IPL is primarily an exercise of translating IJL "structures" to and from the format of the IPL Image structure. We have created three functions that encapsulate the above usage models by providing IPL Image generation from JPEG files:

```
IplImage *CreateImageFromJPEG(char *filename);
```

and allowing a user to write an IPL Image to a JPEG file:

```
bool WriteImageToJPEG(IplImage* , char *filename);
```

2.1 Creating an IPL Image from a JPEG File

The code snippet below illustrates a function that creates an IPL image, allocates space for enough image data to store the entire image, and decompresses a JPEG file to the IPL Image buffer.

This function would be used in place of the IPL function `iplCreateImageHeader(…)`; i.e. it would be followed by call to `iplDeallocate(…)` after completing processing and display of the image.

```
#include <stdlib.h>
#include <stdio.h>
#include "ipl.h"
#include "ijl.h"

////////////////////////////////////////////////////////////////////////
// Create an IPL Image from a JPEG file.
// This function creates a flat (not tile based) IPL Image from a JPEG file.
// iplDeallocate must be called to free the IplImage structure
// and memory created with this function
//

IplImage* CreateImageFromJPEG(
  LPCSTR filename)
{
  BOOL      error = FALSE;
  IJLERR    jerr;
  IplImage* aImage = NULL;
```

```
// allocate the JPEG core properties
JPEG_CORE_PROPERTIES jcprops;

__try
{
  jerr = ijlInit(&jcprops);
  if(IJL_OK != jerr)
  {
    // can't initialize IJLib...
    error = TRUE;
    __leave;
  }

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

  // read image parameters: width, height,...
  jerr = ijlRead(&jcprops,IJL_JFILE_READPARAMS);
  if(IJL_OK != jerr)
  {
    // can't read JPEG image parameters...
    error = TRUE;
    __leave;
  }

  // create the IPL Image header, using a NULL tile info struct.
  aImage = CreateImageHeaderFromIJL(&jcprops,NULL);
  if(NULL == aImage)
  {
    // can't create IPL image header...
    error = TRUE;
    __leave;
  }

  // allocate memory for image
  iplAllocateImage(aImage,0,0);
  if(NULL == aImage->imageData)
  {
    // can't allocate memory for IPL image...
    error = TRUE;
    __leave;
  }

  // tune JPEG decomressor
  jcprops.DIBBytes    = (BYTE*)aImage->imageData;
  jcprops.DIBWidth    = jcprops.JPGWidth;
  jcprops.DIBHeight   = jcprops.JPGHeight;
  jcprops.DIBChannels = 3;
  jcprops.DIBPadBytes = IJL_DIB_PAD_BYTES(jcprops.DIBWidth,
    jcprops.DIBChannels);

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

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

  default:
    {
      jcprops.DIBColor = (IJL_COLOR)IJL_OTHER;
      jcprops.JPGColor = (IJL_COLOR)IJL_OTHER;
      break;
    }
```

```
    }

    // read data from the JPEG into the Image
    jerr = ijlRead(&jcprops,IJL_JFILE_READWHOLEIMAGE);
    if(IJL_OK != jerr)
    {
      // can't read JPEG image data...
      error = TRUE;
      __leave;
    }
  } // try


  __finally
  {
    // release the JPEG core properties
    jerr = ijlFree(&jcprops);
    if(IJL_OK != jerr)
    {
      // can't free IJLib...
      error = TRUE;
    }

    if(FALSE != error)
    {
      if(NULL != aImage)
      {
        iplDeallocate(aImage,IPL_IMAGE_ALL);
        aImage = NULL;
      }
    }
  }

  return aImage;
} // CreateImageFromJPEG()


////////////////////////////////////////////////////////////////////////
// Initalize an IPL Image given a JPEG image
//

IplImage* CreateImageHeaderFromIJL(
  const JPEG_CORE_PROPERTIES* jcprops,
  IplTileInfo*                tileInfo)
{
  int       channels;
  int       alphach;
  char      colorModel[4];
  char      channelSeq[4];
  IplImage* image;

  switch(jcprops->JPGChannels)
  {
  case 1:
    {
      channels      = 3;
      alphach       = 0;
      colorModel[0] = channelSeq[2] = 'R';
      colorModel[1] = channelSeq[1] = 'G';
      colorModel[2] = channelSeq[0] = 'B';
      colorModel[3] = channelSeq[3] = '\0';
      break;
    }

  case 3:
    {
      channels      = 3;
      alphach       = 0;
      colorModel[0] = channelSeq[2] = 'R';
      colorModel[1] = channelSeq[1] = 'G';
      colorModel[2] = channelSeq[0] = 'B';
```

```
      colorModel[3] = channelSeq[3] = '\0';
      break;
    }

  default:
    {
      // number of channels not supported in this samples
      return NULL;
    }
  } // switch

  image = iplCreateImageHeader(
    channels,
    alphach,
    IPL_DEPTH_8U,
    colorModel,
    channelSeq,
    IPL_DATA_ORDER_PIXEL,
    IPL_ORIGIN_TL,
    IPL_ALIGN_DWORD,
    jcprops->JPGWidth,
    jcprops->JPGHeight,
    NULL,
    NULL,
    NULL,
    tileInfo);

  return image;
} // CreateImageHeaderFromIJL()
```

This function is generally applicable but may pose performance problems in certain situations. For example, if a JPEG image's dimensions are too large to fit in available memory, CreateImageFromJPEG will fail. Performance may be poor due to non-local image access.

For these situations we can exploit IJL's random access feature to access the JPEG image as if it were tile based. In one simple implementation of this idea, `CreateTiledImageFromJPEG(…)` creates an IPL Image that accesses information from 128x64 tiles. No decompressed data is stored in the IPL Image; data is accessed dynamically through a callback (`ReadDataFromJPEG(…)`). This usage model dramatically reduces memory use and can result in a significant gain in speed for some operations.

The following code illustrates an implementation of these functions:

```
////////////////////////////////////////////////////////////////////
// The following tile constants are used by the IPL Image
// to segment the image into multiple tiles.  We choose tiles wider than they
// are high because JPEG images are accessed more efficiently reading
// horizontal data than from reading vertical data.

static const TILESIZEX = 128;
static const TILESIZEY = 64;
```

```
//////////////////////////////////////////////////////////////////////
// Create a tile-based IPL Image from a JPEG file.
// After calling this function, call ReleaseIJLForTiledImage
// followed by iplDeallocate to free the JPEG core properties and
// IPL Image structure.
//

IplImage* CreateTiledImageFromJPEG(
  LPCSTR filename)
{
  BOOL                 error;
  IJLERR               jerr;
  BYTE*                buffer  = NULL;
  IplImage*            aImage  = NULL;
  JPEG_CORE_PROPERTIES* jcprops = NULL;

  error = FALSE;

  __try
  {
    // NOTE: freeng jcprops made in ReleaseIJLForTiledImage()
    jcprops = new JPEG_CORE_PROPERTIES;
    if(NULL == jcprops)
    {
      error = TRUE;
      __leave;
    }

    jerr = ijlInit(jcprops);
    if(IJL_OK != jerr)
    {
      // can't initialize IJLib...
      error = TRUE;
      __leave;
    }

    // Open the JPEG
    jcprops->JPGFile = const_cast<LPSTR>(filename);

    jerr = ijlRead(jcprops,IJL_JFILE_READPARAMS);
    if(IJL_OK != jerr)
    {
      // can't read JPEG parameters...
      error = TRUE;
      __leave;
    }

    // create the IPL tileInfo.
    IplTileInfo* tileInfo = iplCreateTileInfo(
      &ReadDataFromJPEG,
      (void*)jcprops,
      TILESIZEX, TILESIZEY);
    if(NULL == tileInfo)
    {
      // can't allocate memory for tileInfo...
      error = TRUE;
      __leave;
    }

    aImage = CreateImageHeaderFromIJL(jcprops,tileInfo);
    if(NULL == aImage)
    {
      // can't create IPL image header...
      error = TRUE;
      __leave;
    }

    buffer = new BYTE [TILESIZEX * TILESIZEY * 3];
    if(NULL == buffer)
    {
      // can't allocate memory for buffer...
```

```
      error = TRUE;
      __leave;
    }

    // tune JPEG decomressor
    jcprops->DIBBytes    = buffer;
    jcprops->DIBWidth    = TILESIZEX;
    jcprops->DIBHeight   = TILESIZEY;
    jcprops->DIBChannels = 3;
    jcprops->DIBPadBytes = IJL_DIB_PAD_BYTES(jcprops->DIBWidth,
      jcprops->DIBChannels);

    switch(jcprops->JPGChannels)
    {
    case 1:
      jcprops->JPGColor = IJL_G;
      break;

    case 3:
      jcprops->JPGColor = IJL_YCBCR;
      break;

    default:
      jcprops->DIBColor = (IJL_COLOR)IJL_OTHER;
      jcprops->JPGColor = (IJL_COLOR)IJL_OTHER;
      break;
    }
  } // __try

    __finally
  {
    if(FALSE != error)
    {
      if(NULL != aImage)
      {
        ReleaseIJLForTiledImage(aImage);
        iplDeallocate(aImage,IPL_IMAGE_ALL);
      }
    }
  }

  return aImage;
} // CreateTiledImageFromJPEG()


////////////////////////////////////////////////////////////////////////
// Release the JPEG core properties structure allocated
// by CreateTiledImageFromJPEG (above).
//

void ReleaseIJLForTiledImage(
  IplImage* aImage)
{
  JPEG_CORE_PROPERTIES* jcprops = (JPEG_CORE_PROPERTIES*)aImage->tileInfo->id;

  if(NULL != jcprops)
  {
    if(NULL != jcprops->DIBBytes)
    {
      delete [] jcprops->DIBBytes;
    }

    ijlFree(jcprops);

    // NOTE: jcprops is allocated in CreateTiledImageFromJPEG()
    delete jcprops;
    jcprops = NULL;
  }

  return;
} // ReleaseIJLForTiledImage()
```

```
//////////////////////////////////////////////////////////////////////
// Callback function for access to IPLImage by tiles
//

void __stdcall ReadDataFromJPEG(
  const IplImage* image,
  int             xIndex,
  int             yIndex,
  int             mode)
{
  IJLERR jerr;

  // we don't allow writing by roi.
  if(IPL_GET_TILE_TO_WRITE == mode || IPL_RELEASE_TILE == mode)
    return;

  JPEG_CORE_PROPERTIES* jcprops = (JPEG_CORE_PROPERTIES*)image->tileInfo->id;

  jcprops->jprops.roi.left   = xIndex * TILESIZEX;
  jcprops->jprops.roi.top    = yIndex * TILESIZEY;
  jcprops->jprops.roi.right  = (xIndex + 1) * TILESIZEX;
  jcprops->jprops.roi.bottom = (yIndex + 1) * TILESIZEY;

  jerr = ijlRead(jcprops,IJL_JFILE_READWHOLEIMAGE);
  if(jerr < IJL_OK)
  {
    // can't read from JPEG
  }

  // data is to be written here:
  image->tileInfo->tileData = (char*)jcprops->DIBBytes;

  return;
} // ReadDataFromJPEG()
```

2.2 Writing a JPEG file from an IPL Image.

The following code snippets illustrate how to author JPEG images from IPL Images. As IJL can only create JPEG images from a planar data source, the example we give is limited to IPL Images that

1. Contain image data (no tiled IPL Images)

2. Contain data in a format useful to IJL (pixel oriented, 8-bit images)

It would be easy to extend this exercise to include formatting functions designed to format any given IPL image into a form suitable for IJL.

```
//////////////////////////////////////////////////////////////////////
// Write an IPLImage to a JPEG file.
// This function requires an IPL Image that is not tile based
//

bool WriteImageToJPEG(
  IplImage* aImage,
  LPCSTR   filename)
{
  bool                bres;
  IJLERR              jerr;
  JPEG_CORE_PROPERTIES jcprops;

  bres = true;
```

```
  __try
  {
    jerr = ijlInit(&jcprops);
    if(IJL_OK != jerr)
    {
      // can't initialize IJLib...
      bres = false;
      __leave;
    }

    bres = SetJPEGProperties(jcprops,aImage);
    if(false == bres)
    {
      __leave;
    }

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

    jerr = ijlWrite(&jcprops,IJL_JFILE_WRITEWHOLEIMAGE);
    if(IJL_OK != jerr)
    {
      // can't write JPEG image...
      bres = false;
      __leave;
    }

  } // __try


  __finally
  {
    ijlFree(&jcprops);
  }

  return bres;
} // WriteImageToJPEG()


////////////////////////////////////////////////////////////////////////
// Set JPEG properties from IplImage
//

#define IS_RGB(image) \
  ( image->colorModel[0] == 'R' && \
    image->colorModel[1] == 'G' && \
    image->colorModel[2] == 'B' )

#define IS_GRAY(image) \
  ( image->colorModel[0] == 'G' && \
    image->colorModel[1] == 'R' && \
    image->colorModel[2] == 'A' && \
    image->colorModel[3] == 'Y' )

#define IS_SEQUENCE_RGB(image) \
  ( image->channelSeq[0] == 'R' && \
    image->channelSeq[1] == 'G' && \
    image->channelSeq[2] == 'B' )

#define IS_SEQUENCE_BGR(image) \
  ( image->channelSeq[0] == 'B' && \
    image->channelSeq[1] == 'G' && \
    image->channelSeq[2] == 'R' )


bool SetJPEGProperties(
  JPEG_CORE_PROPERTIES& jcprops,
  const IplImage*       image)
{
  if(IPL_DEPTH_8U != image->depth)
  {
    // only IPL_DEPTH_8U is supported now
```

```
  return false;
}

if(IPL_DATA_ORDER_PIXEL != image->dataOrder)
{
  // only IPL_DATA_ORDER_PIXEL is supported now
  return false;
}

if(!IS_RGB(image) && !IS_GRAY(image))
{
  // only RGB or GRAY color model supported in this sample
  return false;
}

if(image->nChannels != 1 && image->nChannels != 3)
{
  // only 1 or 3 channels supported in this example
  return false;
}


jcprops.DIBChannels = image->nChannels;

// set the color space
// assume that 1 channel image is GRAY, and
// 3 channel image is RGB or BGR
switch(jcprops.DIBChannels)
{
case 1:
  jcprops.DIBColor    = IJL_G;
  jcprops.JPGColor    = IJL_G;
  jcprops.JPGChannels = 1;
  break;

case 3:
  if(IS_SEQUENCE_RGB(image))
  {
    jcprops.DIBColor    = IJL_RGB;
    jcprops.JPGColor    = IJL_YCBCR;
    jcprops.JPGChannels = 3;
  }
  else if(IS_SEQUENCE_BGR(image))
  {
    jcprops.DIBColor    = IJL_BGR;
    jcprops.JPGColor    = IJL_YCBCR;
    jcprops.JPGChannels = 3;
  }
  else
  {
    jcprops.DIBColor    = (IJL_COLOR)IJL_OTHER;
    jcprops.JPGColor    = (IJL_COLOR)IJL_OTHER;
    jcprops.JPGChannels = 3;
  }
  break;

default:
  // error for now
  break;
}

jcprops.DIBHeight      = image->height;
jcprops.DIBWidth       = image->width;
jcprops.JPGHeight      = image->height;
jcprops.JPGWidth       = image->width;
jcprops.JPGSubsampling = (IJL_JPGSUBSAMPLING)IJL_NONE;

if(IPL_ORIGIN_BL == image->origin)
{
  jcprops.DIBHeight = -jcprops.DIBHeight;
}
```

```
  switch(image->align)
  {
  case IPL_ALIGN_4BYTES:
    jcprops.DIBPadBytes = IJL_DIB_PAD_BYTES(jcprops.DIBWidth,
      jcprops.DIBChannels);
    break;

  case IPL_ALIGN_8BYTES:
    jcprops.DIBPadBytes = (jcprops.DIBWidth*jcprops.DIBChannels + 7)/8*8 -
      jcprops.DIBWidth*jcprops.DIBChannels;
    break;

  case IPL_ALIGN_16BYTES:
    jcprops.DIBPadBytes = (jcprops.DIBWidth*jcprops.DIBChannels + 15)/16*16 -
      jcprops.DIBWidth*jcprops.DIBChannels;
    break;

  case IPL_ALIGN_32BYTES:
    jcprops.DIBPadBytes = (jcprops.DIBWidth*jcprops.DIBChannels + 31)/32*32 -
      jcprops.DIBWidth*jcprops.DIBChannels;
    break;

  default:
    // error if go there
    break;
  }

  // set the source for the input data
  // note-this will fail if the image is tile based.
  jcprops.DIBBytes = (BYTE*)image->imageData;

  return true;
} // SetJPEGProperties()
```