Gamma correction of RGB images using TI DSPs

Shehrzad Qureshi, author, *Embedded Image Processing on the TMS320C6000TM DSP* - June 17, 2005

In my new book *Embedded Image Processing on the TMS320C6000TM DSP: Examples in Code Composer Studio™ and MATLAB*, which can
Editor's Note: Shehrzad Qureshi's new book, *Embedded Image Processing on the TMS320C6000™ DSP : Examples in Code Composer Studio™ and MATLAB*, explains how to implement a variety of common image processing algorithms on Texas Instruments (TI) floating and fixed point Digital Signal Processors (DSPs). More info on his book can be found at the end of this original new article about achieving gamma correction with TI DSPs -- a topic not included in the book.

In this article, I will discuss a very common color imaging algorithm known as gamma correction, and explain how to build a host C# application that uses modern, flexible, and easy-to-use .NET facilities to read in a color image, package it up, and send it down to the DSP. The .NET C# host application is capable of communicating with a DSP development platform *in this case, the C6416 DSP Development Kit (DSK), using the Real-Time Data Exchange™ (RTDX) data transfer protocol*. I will explain a simple, unoptimized, DSP program that performs the grunt work, and provide pointers into my book on how to then optimize this program. I conclude the article with an "open letter" to the DSP community with some questions on how to improve the RTDX data transfer.

This article assumes some familiarity with CCStudio versions 2.20 or 3.0, as well as the Visual Studio Studio .NET IDE. All source code for this article can be found on my web site, [www.squareshi.com](http://www.squareshi.com). Note that this site contains periodic updates for the book, including errata. In fact, an update was posted just recently that describes how a reader altered the projects presented in Chapter 4 in order to get the code to run on the DM642 Evaluation Module (EVM), a TI development platform not covered in the book.
Gamma Correction

Gray-scale transform functions are implemented most efficiently via look-up tables (LUTs). In general, there are three commonly used categories of these LUTs:

1. linear
2. piece-wise linear
3. non-linear

Gamma correction LUTs fall under (3), and provide a means of enhancing the quality of an image. The gamma correction transfer function is a power function that compensates for the nonlinear relationship between pixel intensities and the intensities of a display device. See [1] and [2] for more background on the rationale and motivation behind employing gamma correction.

Construction of the LUT used for implementing gamma correction is straightforward, given the gamma parameter, g. The transfer function remaps pixel intensities according to the following relation:

\[ \text{outpixel} = \left[\text{inpixel}^g\right] \]
The floor operator is present because when performing gamma correction on digital devices we must quantize the output in order to fit the output pixel intensity into a fixed-length quantity (typically an 8-bit unsigned integer). A few examples of gamma transfer functions are shown in Figure 1.

![Figure 1](image)

**Figure 1**: Three examples of gamma correction LUTs.

As previously stated, *Embedded Image Processing on the TMS320C6000 DSP* restricts its focus to gray-scale or monochrome images, sans a single example illustrating how to perform histogram equalization on an RGB color image in MATLAB (see section 3.4.2). Gamma correcting an RGB color image is conceptually simpler than histogram equalizing a color image, as we merely process each RGB color plane, or channel, separately by passing each of them through the gamma correction LUT. In essence, we treat each of the three RGB planes as its own monochrome image. This is in contrast to the histogram equalization case where we must first perform an RGB-to-HSV (Hue, Saturation, Value) color transformation, modify just the value plane, and then finally perform the reverse (HSV-t-RGB) transform. A high-level depiction of RGB color image gamma correction is shown in Figure 2.
Another motif in the book is the philosophy regarding MATLAB prototypes. In general, because the book covers fairly standard image processing primitives and algorithms, many of these can be utilized quite readily within MATLAB through the use of various toolboxes, the most germane to this discussion being the Image Processing Toolbox\[3\]. However, simply referring the reader to a toolbox function is not very helpful, as then these functions tend to be viewed as black-boxes that are not as useful when considering the ultimate goal, that of getting efficient C code running on the DSP. To that end, in almost every case I include a hand-coded replacement M-file, which is more than likely to not be as full-featured as the MATLAB version, but which is more helpful to the reader in understanding the actual implementation of the algorithm. As one might expect, I do the same here with respect to gamma correction.

Performing gamma correction is very easy in MATLAB if the Image Processing Toolbox is available. As described in [4], the imadjust Image Processing Toolbox function can be used to perform a number of intensity transformation operations, and one of its arguments is a gamma parameter that shapes the intensity transform curve a la Figure 1. Gamma correction is performed by modifying this final parameter, as illustrated in Figure 3.
Figure 3: Using imadjust to do gamma correction in MATLAB.

The empty brackets in the imadjust calls are placeholders for MATLAB, indicating that we wish to modulate the entire pixel intensity range, as opposed to a subset (which would make our nonlinear transform a piecewise nonlinear transform). In the source code distribution accompanying this article, I have included an M-file gamma_correct.m that implements gamma correction on MATLAB matrices of type double, uint8, or uint16. This function also returns as an optional second output argument the actual LUT used to transform the pixel intensities. If the matrix contains double elements, this function assumes that pixel values have been scaled to the range [0-1.0]. Likewise, if the data type is uint8 or uint16, gamma_correct assumes a pixel range of [0-255] or [0-65535], respectively. This M-file (below), utilizes a few MATLAB tricks to efficiently vectorize the LUT transformation, which are fully explained in section 3.3.2 of my book.

M-file:

```matlab
function [J, varargout] = gamma_correct(I, gamma)

% GAMMA_CORRECT applies gamma LUT to input image
```
% J = GAMMA_CORRECT(I, gamma) or [J, LUT] = 
% GAMMA_CORRECT(I, gamma)
%
% J: output image, of the same type as I
% I: input image
% LUT: gamma correction lookup-table (transform function)
%
max_intensity = max_pix_val(I);

% create lookup table based on gamma (must take care to scale
% table elements correctly.
LUT = max_intensity .* (num2cell([0:max_intensity]./max_intensity).^gamma);
LUT = floor(LUT); % floor instead of round to simulate C code

% apply LUT to all image planes - here I utilize a MATLAB
% 'trick'
% to elegantly apply the indexing operation (the +1 is because
% MATLAB is ones-based!)
J = LUT(double(I)+1); % convert I to double b/c MATLAB only
% vectorized indexing on double types
J = cast(J, class(I));

% return the transform function if the user asked for it.
if 2 == nargout
    varargout{1} = LUT;
end

function v = max_pix_val(I)
%
% A little helper sub-function that given an image
% matrix I, returns the maximum pixel value based on
% I's class.
%
if isa(I, 'double') | isa(I, 'single')
    v = 1.0;
elseif isa(I, 'uint8')
    v = 255;
elseif isa(I, 'uint16')
    v = 65535;
else
    error(sprintf('unsupported class type %s', class(I)));
end

Gamma Correction of RGB images on the C6416 DSP

The source code accompanying this article contains a program developed and tested using version 2.20 of CCStudio, targeting the fixed-point C6416 DSP. This program differs from those in my book in two facets. First, this C code processes RGB images, which it stores as three separate one-dimensional flattened arrays. Secondly, whereas the DSP programs in *Embedded Image Processing on the TMS320C6000 DSP* deal with fixed (at compile time) dimension images, this program is flexible enough to handle images whose dimensions may vary during run time, although 8-bit color planes are assumed. The project consists of the following files, all located in the Target directory:

- `imgproc_rtdx.c` : the main C source file
- `imgproc_rtdx.cmd` : linker command file
- `imgproc_rtdx.pjt` : CCStudio v2.2.0 project file
- `intvecs6416.asm` : interrupt vector table, needed for initialization
- `rtdx_buf.c` : RTDX buffer size configuration
The CCStudio project file will more than likely need to be modified to suit the reader's particular setup. For example, on the author's PC the base CCStudio directory is C:\TIC6416DSK; on the reader's machine it may be C:\TI, and hence one would need to change imgproc_rtdx.pjt file appropriately. The general flow of the program is as follows:

1. Wait for image dimensions from host.
2. Read gamma value and RGB color data from host.
3. Create gamma correction LUT, based on (1).
4. For each color-plane, perform in-place pixel remapping according to LUT contents.
5. Send processed RGB data back to host.
6. Go to 1.

1, 2, and 5 utilize RTDX and shall be discussed in more depth in subsequent sections. Pointers to the three flattened arrays that store the red, green, and blue color planes are stored in the RGB array, and steps 3 and 4 are implemented within the gamma_correction function (see below). As you can see, because this is an unoptimized DSP implementation (for example, on-chip memory is not being properly utilized), the structure of the source code that performs the actual image processing is quite simple. In fact, most of this particular program's source code deals with data transfer issues. If this was a fully optimized implementation, then this would not be the case, as we would then be incorporating Direct Memory Access (DMA) into the process, which definitely complicates matters.

```c
void gamma_correction(int num_rows, int num_cols)
{
    int ii=0, num_pixels = num_rows*num_cols;
    unsigned char *pr=RGB[0], *pg=RGB[1], *pb=RGB[2];
```
/* create LUT */
for(; ii

An Interactive Host/Target Gamma Correction App Using RTDX and C#

RTDX is a TI communications protocol that allows one to build applications where a host program, residing on a PC, talks to a target DSP program via Code Composer Studio. Section 5.1.2 of my book gives an introduction to RTDX, and two different types of RTDX-enabled host applications are discussed and provided on the book CD-ROM. In Chapter 5 a MATLAB application, which actually communicates with Code Composer Studio using a Mathworks product Link for Code Composer Studio, is given. In Chapter 6, similar functionality is provided but this time within the context of a C/C++ application developed using Visual Studio .NET 2003. In both of these cases, the host side of RTDX eventually boils down to interfacing with the TI-provided RTDX host library, which takes the form of a COM interface. The target utilizes RTDX in a more traditional fashion, by calling various functions from C defined in the rtdx.h header file.

What you will find is that aside from the code presented in my book, most of the example RTDX host applications use rather dated languages (e.g. Visual Basic 6), and furthermore, most of them demonstrate the transfer of
relatively small amounts of data and none of them deal with images, per se. The host application accompanying this article, in the directory appropriately named Host, contains a modern Windows Forms application written in C# that deals with the image transfer and display issues. The .NET framework provides a multitude of COM interoperability facilities, making it relatively easy to access COM interfaces like the RTDX host object. Nevertheless, there are a number of subtle issues involved and therefore having fully debugged source code provides a wonderful starting point for future development.

On the target side, nothing much changes from the code discussed in my book, except for the fact that here I provide a CCStudio project, in the Target directory, that does not require DSP/BIOS. Before proceeding with a discussion of the source code, an example of how to use the application and a high-level description of the communications protocol is necessary for complete understanding. To run the application, begin by starting CCStudio, but do not load the .out file and run the program just yet. Prior to running the program, it is essential to first configure RTDX (if necessary) and enable RTDX, as by default it is not enabled. To do this, bring up the RTDX configuration control screen using the Tools|RTDX|Configuration Control menu selection. You can change the host buffer size by clicking on the Configure button, but you need only do this if you encounter the following message when running the application:

"A data message was received which cannot fit into the buffer allocated on the host. To avoid data loss reconfigure the buffer and run the application again."

For more detailed information, refer to the "Reconfiguring the Host Buffer" topic in the CCStudio online help (see the RTDX FAQ). For this application, a host buffer size of 921600 (640*480*3) is sufficient, given the maximum
image size the target currently supports (you can change this maximum dimension by modifying the MAX_NUM_ROWS and MAX_NUM_COLS preprocessor definitions in imgproc_rtdx.c). Click on the checkbox labeled "Enable RTDX", load the .out file and run the program. You should then see the strings:

Input & Output channels enabled

Waiting for image size from host

Now run the host application by double-clicking on the DotNET_ImageProc_RTDX.exe application. You can load in an image file via the File|Load menu selection, and alter the gamma value using the slider bar. Clicking on the button labeled "Process" will send the RGB data down to the target, and upon receipt of the processed data, the gamma corrected image will be rendered in the right-hand pane. During the actual image processing, you can monitor the progress of the target by looking at stdout within CCStudio. Figure 4 is a screen-capture of the host application, after having loaded in the same image from Figure 3 and passed it through the gamma correction program running on the C6416 DSK.
As discussed in 5.1.2 of my book, the writing of large blocks of data from the host to the target is far less problematic than the converse, reading of large blocks of data emanating from the target on the host. Each of the red, green, and blue channels is dealt with separately, both in a processing as well as image transfer perspective. From the host's vantage point, the sending of an image can be encapsulated in a single RTDX message. However, the target is not able to send that same amount of data back to the host. As a consequence, sending back the processed image data is somewhat of a chore, and the color channel is segmented into 1K blocks and sent one by one. Since the color channels are not necessarily an even multiple of 1K, a final "remainder" block may be sent to complete each color channel transfer. Figure 5 depicts the data transfer protocol.
Figure 5: Data Transfer Protocol between C# host and DSP target.

Given the protocol description and Figure 5 in particular, the remainder of the target DSP C implementation will become clearer. While RTDX is a full-duplex protocol, each RTDX channel is half-duplex, meaning that to implement the above protocol both the host and target require two RTDX objects. In the case of the target C program, the "object" takes the form of two opaque data types referenced by the ichan and ochan variables. The reading of a single element, which is done to extract the gamma value and image size (the number of rows and number of columns are packed into the low and high portions of a 32-bit integer), is accomplished by calling the RTDX_read function. The same function is also used to read in an array of data, which is utilized to pull in each of the RGB color channels from the host. Listing 3 gives the functions read_RGB_from_host and
write_RGB_to_host from imgproc_rtdx.c, where read_RGB_from_host implements part of the top of Figure 5 and write_RGB_to_host packages each of the three color channels into separate packets and ships them off to the target.

/* read color planes via RTDX, return 1 if successful, 0 otherwise */
int read_RGB_from_host(int nr, int nc)
{
    int num_pixels = nr*nc, status, ii=0;
    for (; ii

/* write processed color planes back to host (1==success, else 0) */
int write_RGB_to_host(int nr, int nc)
{
    /*
     * 1st thing we must do is tell the host how we're going to split up
     * the image frames (if we need to), since in general, we cannot
     * assume that the RTDX write buffer will be able to handle an entire
     * frame with a single RTDX_write call.
     */
    const unsigned int PKT_SIZE_BYTES = 1024;
    int num_pixels = nr*nc,
    num_packets = num_pixels/ PKT_SIZE_BYTES,
    remainder_bytes = (num_pixels -
    (PKT_SIZE_BYTES*num_packets)),
    remainder_packet = (0 != remainder_bytes),
    ii=0, jj;
    unsigned char *pRGB = 0;

    /*
     * send msg to host telling it what the packet size is,
     * it will then discern the # of packets.
     */
    if (!RTDX_write(&ochan,
        (void *)& PKT_SIZE_BYTES,
        sizeof(PKT_SIZE_BYTES))) {
        printf("ERROR: RTDX write of packet size failed!\n");
As far as Windows Forms applications go, the C# application that constitutes the host application is rather simple, even though it includes a fair amount of functionality. The overall "richness" of this application attests to the expressive power of the .NET framework, in particular the rendering of 2D image data and the file I/O associated with reading in standard image formats such as JPEG and Windows Bitmap (.bmp) files. The GDI+ library, which is an integral part of .NET, is used heavily in this application, as it also is on the C/C++ host applications that are discussed in *Embedded Image Processing on the TMS320C6000 DSP*. In the book, I provide a C++ wrapper class that simplifies GDI+ even further when used to process 8-bit monochrome images and provides some amount of integration with the [Intel Integrated Performance Primitive library](https://software.intel.com/en-us/intel-ipp).

In order to use RTDX in a Visual Studio project, one must initially add a reference to the RTDX host-side COM object into the project. To do this, switch over to the solution view, right-click on the References folder, and select the "Add Reference" menu item. When the ensuing dialog appears, tab over to the "COM" tab, which enumerates all COM objects registered on the system. Click on the Browse button and traverse to [TI]\cc\bin, where [TI] references the CCStudio installation directory (e.g. C:\TI), and double-click on the DLL named rtdxint.dll. The component should now
show up at the bottom of the Add Reference dialog (shown below in Figure 6), and click on the OK button to add a reference to the TI RTDX COM component to the .NET project. With the type library now inserted into your project, it is now possible to instantiate the RTDX component, which is done in frmMain.cs via the following lines of code:

```csharp
private RTDXINTLib.RtdxExpClass rRTDX = new RTDXINTLib.RtdxExpClass(),

wRTDX = new RTDXINTLib.RtdxExpClass();
```

![Figure 6: Inserting the RTDX COM type library into a .NET project.](image-url)
Just as in the target, we need two RTDX objects to read and write data, except here the RTDX channels take the form of COM objects. These two RTDX channels are initialized in the form's constructor.

The images are drawn to .NET PictureBox controls. The act of displaying an image becomes borderline trivial, as simply binding a GDI+ image object to the PictureBox, via it's Image property, suffices to have Windows Forms render the bitmap pixels with each screen update. This is performed within the callback (or delegate, to use .NET terminology) for the **File|Load** menu selection, the `mainMenuFileLoad_Click` method:

```csharp
this.inImgPixBox.Image = Image.FromFile(dlg.FileName);
```

where `dlg.FileName` is the fully-qualified path to the image the user chose and `inImgPixBox` is a class variable pointing to the left-hand side of the main form (see Figure 4). One thing to keep in mind when developing similar forms in the future is that a PictureBox's SizeMode attribute is an essential parameter that should be set at design time. In this project, both PictureBoxes SizeMode attributes are set to StretchImage, meaning which no matter the size of the image the user loads into the application, when it is painted into the PictureBox it will either expand or shrink to fit exactly within the confines of the control. By default, SizeMode is not set to StretchImage, and as a result images will stand a good chance of being cropped when they are drawn.
The RTDX COM interface, built as it is to shuttle data between the host PC and CCStudio, contains various methods that accept arrays of pre-allocated data. This means that if one is developing in so-called "native" or "unmanaged" C++ (without .NET), then one must be familiar with a COM construct known as the SAFEARRAY. In general, COM programming is quite verbose in C/C++, and especially so with SAFEARRAYs. Thankfully much of this headache goes away in C# due to the very convenient "COM interop" features that .NET provides, which is one of the definite benefits of using this language to develop RTDX-aware host applications. Once one understands some of the intricacies behind COM interop, Microsoft's technology for bridging the gap between legacy COM objects like the RTDX host library and the newer .NET framework, the source code begins to fall into place.

Listing 4 (following the end of this article -- see below) contains the portion of frmMain.cs that gets executed when the user clicks on the Process button. One key point concerns obtaining pointers into GDI+ objects and other "managed" arrays that contain pixel data. In C/C++, this point is moot because the language obviously supports pointers, however this most definitely is not the case in the .NET world. In C#, we must bracket the code that uses C-style pointer accesses with the unsafe keyword, but we then gain low level C-style byte-addressable pointer accesses which allow us to both package the color plane pixel data for transmission as well as marshal the processed data coming back from the target into a GDI+ object encapsulating a bitmap. Finally, note that we need not deal explicitly with COM constructs like the SAFEARRAY or variant. We can simply pass a managed array of bytes to the RTDX Write method (see the sendColorPlane method), and COM interop then handles all the rest. Likewise, when reading the individual color planes back from the target, the data arrives packaged into a System.Array object, which we then proceed to marshal into the GDI+ bitmap structure encapsulating the gamma corrected RGB image (see readColorPlane method, and the child methods that it calls).
Ideas for Improvement

This application feels slower than it should be. One of the major bottlenecks is reading the processed RGB color data back from the DSK, as we are forced to split the image into 1K packets which in turn greatly increases our transmission overhead. In addition, the host software does not employ an event-based model and instead simply keeps attempting to read the next set of data within a tight loop. The end result is that the host CPU usage ramps up and the GUI essentially hangs while it waits for the data from the target, which in this case is somewhat acceptable since it is a simplistic demonstration application. A better design however, would be to at the very least spin the host-side RTDX reads off into their own thread, instead of falling back on the polling paradigm used here. Ideally, one would prefer perhaps utilizing some of the other RTDX host library methods to implement an event-based target-to-host read mechanism, thereby mitigating the need for direct polling in the host application.

On the DSP side, the gamma correction code is largely unoptimized. A huge speedup will be obtained by caching portions of the image in on-chip RAM, prior to application of the gamma LUT, via the EDMA controller present on the C6416. For more information on this DMA optimization, refer to Chapter 4 of Embedded Image Processing on the TMS320C6000 DSP.

A Request for Help

I was obviously aware that the target-to-host image transfer constitutes a serious bottleneck, however try as I may I was unable to transmit large blocks of 8-bit data reliably from the target to the host using the RTDX_write API function. Some of what I tried included:
• Increasing the host buffer size in the RTDX configuration dialog, so that CCStudio never complained about overrunning the buffer.
• Using RTDX continuous mode, where the use of the .rtd file is eschewed and CCStudio transfers data through the host presumably over shared memory. The default is non-continuous mode, where CCStudio transfers data using (by default) a file named logfile.rtd.
• Increasing the target buffer size by increasing the value of the preprocessor symbol BUFRSZ in rtdx_buf.c, as described in the CCStudio online help topic "How do I change the size of the RTDX buffer on the target?" (see RTDX FAQ section).

Alas, none of the above worked. Ideally, one would prefer to be able to send the entire color plane using a single RTDX message, just as in the opposite direction, but what was discovered was that RTDX would be able to handle one large write message (say on the order of 307,200 bytes) but then successive calls would mysteriously fail, without any feedback whatsoever. My hope is that some of the TI DSP experts out there will read this and point me in the right direction. Perhaps rewriting the target program to incorporate the DSP/BIOS real-time operating system, and using the HST module in lieu of the RTDX API library, works better with relatively large RTDX messages?

**Conclusion**

Regardless of the issues described above, in this article I presented a fully featured color image processing application with a modern C# front-end and embedded DSP back-end, which does much of the heavy lifting. RGB images, of varying sizes, are transferred over RTDX channels to be gamma corrected by the DSP and read back and displayed on the host PC. In the course of this discussion, a MATLAB prototype that implements some of what the Image Processing Toolbox imadjust function was given, and some
advanced C# techniques that allow efficient low-level access to GDI+.NET objects are present within the source code. All source code and sample image data can be downloaded from


**Editor's Footnote:**

Shehrzad Qureshi is the author of a new book, *Embedded Image Processing on the TMS320C6000 DSP (Examples in Code Composer Studio and MATLAB)*. The book covers a variety of common image processing algorithms and illustrate how to implement them in an efficient manner on Texas Instruments (TI) floating and fixed point Digital Signal Processors (DSPs). The book utilizes a "cookbook" expository style, where each algorithm discussion includes:

1. Introduction to the algorithm in its theoretical context.
2. MATLAB prototype(s), in the form of M-files
3. C/C++ "first-cut" prototype implementation, using the Microsoft Visual Studio.NET 2003 IDE.

The book provides readers with a number of templates, or test-bench frameworks, upon which one can easily graft future imaging applications. For example, on the accompanying book CD-ROM, there are a number of host (PC) applications designed to communicate to a target (DSP) program, where the host application, be it a MATLAB or Visual Studio C/C++ application, is responsible for file I/O, data transfer, user input, and visualization duties, whereas the program running on the DSP target performs the actual image processing as well as its end of the data transfer
pipe. Thus one is free to alter the "back-end" target executable to suite his/her purposes, thereby making it far easier to develop a fully functional image processing test-bench application, replete with all the necessary plumbing.

More information can be obtained at the author's web site, www.squareshi.com, or at the publisher's web site.

References


Listing 4 (see article):

/// The main work-horse method in this application. It sends the input
/// image down to the target DSP. It then reads the processed image
/// back from the target. Finally it marshals this raw pixel data into
/// something .NET palatable.
private void processButton_Click(object sender, System.EventArgs e)
{
    if (null != this.inImgPixBox.Image)
    {
        this.Cursor = Cursors.WaitCursor;
        this.sendImgDimsAndGamma();

        // processed image will go into this picture box
        this.gammaImgPixBox.Image = (Image)this.inImgPixBox.Image.Clone();

        // send RGB color planes to target
        Bitmap bm = (Bitmap)this.gammaImgPixBox.Image;
        BitmapData bmData = null;
        try
        {
            bmData = bm.LockBits(new Rectangle(0, 0, bm.Width, bm.Height),
                ImageLockMode.ReadWrite,
                PixelFormat.Format24bppRgb);
            int stride = bmData.Stride;
            System.IntPtr scan0 = bmData.Scan0;
            this.sendImgData(scan0, bm.Width, bm.Height, stride);

            // target should now be processing the image, here we keep
            // polling for the incoming packet size - upon receiving this msg
            // from the target we proceed to read the actual image
frame
  // data.
  int nTotalBytesPerPlane = bm.Width * bm.Height,
     nRemainderPacketBytes,
     nPacketSizeBytes;
  int nPacketsPerPlane
     = this.wait4PktSize(out nPacketSizeBytes,
                        out nRemainderPacketBytes,
                        nTotalBytesPerPlane);

  // now start reading processed data back from the DSP
  for (int plane = 0; plane

  /// 1st sends the image height (# rows) and width (# cols) in
  /// a single
  /// RTDX write msg, and then sends the gamma parameter in a 2nd
  /// RTDX
  /// write msg.
  private void sendImgDimsAndGamma()
  {
    // encode and then send image dimensions to target
    int sz = (this.inImgPixBox.Image.Height

    // is the image too big?
    int bOk = 0, nHowMany = 0;
    while (0 != this.readRTDX.ReadI4(out bOk))
    {
      if (nHowMany > 10)
        throw new Exception("RTDX read of target confirmation
failed.");
      nHowMany++;
    }
    if (0 == bOk)
      throw new Exception("Image dimensions too large");

    // send gamma parameter to DSP target
    if (0 !=
        this.writeRTDX.WriteF4(float.Parse(this.gammaTextBox.Text),
                               out nHowMany))
throw new Exception("RTDX write of gamma failed.");
}

/// This method is responsible for sending each of the 3 RGB
/// planes down to the target.
private void sendImgData(System.IntPtr scan0,
                        int width, int height, int stride)
{
    // see http://www.codeproject.com/cs/media/csharpgraphicfilters11.asp
    byte [] redPlane = new byte[width*height],
            greenPlane = new byte[width*height],
            bluePlane = new byte[width*height];
    unsafe {
        // 1st copy channels into 3 separate flattened arrays
        byte *p = (byte *)(void *)scan0;
        int nOffset = stride - width*3, ii=0;
        byte red, green, blue;

        for(int y=0; y

            redPlane[ii] = red;
            greenPlane[ii] = green;
            bluePlane[ii] = blue;
            p += 3;
        }
        p += nOffset;
    }

    // now that we've marshalled the 2D image data into a
    series // of flattened image arrays, send them off to the target
    this.sendColorPlane(redPlane);
    this.sendColorPlane(greenPlane);
    this.sendColorPlane(bluePlane);
} // end unsafe block
}
/// Responsible for sending a single color plane down to the
target.
private void sendColorPlane(byte [] p)
{
    const int N_ATTEMPTS = 10;
    int iAttempt = 0, nBytes;
    while (iAttempt++

    /// A blocking method that continually attempts to read an
    /// acknowledgement msg from the target, which contains the
    /// size of each
    /// packet that the host shall read. Armed with this
    /// information, the
    /// host app is then able to derive a bunch of other data
    /// communication
    /// parameters used when reading the processed color pixels.
    /// All of
    /// this jazz is needed because in contrast to the sending of
    /// pixel
    /// data, we cannot read an entire plane with a single RTDX
    /// msg (we are
    /// forced to split it up into separate packets).
    private int wait4PktSize(out int nPacketSizeBytes,
        out int nRemainderPacketBytes,
        int nTotalBytes)
    {
        while (0 != readRTDX.ReadI4(out nPacketSizeBytes))
            Trace.Write("Failed to read packet size - will attempt
again...\n");
        int nPacketsPerPlane = nTotalBytes / nPacketSizeBytes;
        nRemainderPacketBytes = nTotalBytes -
            nPacketsPerPlane*nPacketSizeBytes;

        Trace.WriteLine("Packet size is " +
            nPacketSizeBytes.ToString());
        return nPacketsPerPlane;
    }

    /// Reads a single color plane, and returns a managed array of
    /// this
    /// pixel data.
private byte[] readColorPlane(int nTotalBytes,
    int nPacketsPerPlane,
    int nPacketSizeBytes,
    int nRemainderPacketBytes)
{
    byte[] colorPlane = new byte[nTotalBytes];
    object sa = new object(); // sa = "safe array"
    int status, kk = 0;
    unsafe
    {
        for (int iPacket = 0; iPacket

            // once we've got it, we need to marshal the packed
            // integers into
            // unsigned bytes
            kk = this.unpackPixels((System.Array)sa,
                nPacketSizeBytes,
                colorPlane, kk);
    }
    if (0 != nRemainderPacketBytes) // handle last packet if necessary
    {
        while (0 != readRTDX.ReadSAI4(out sa))
            Trace.Write("Failed to read final img plane packet" +
                " - will attempt again...
"
            );
            this.unpackPixels((System.Array)sa,
                nRemainderPacketBytes,
                colorPlane, kk);
    }
    return colorPlane;
}

/// RTDX read messages come in as 4-byte integers, here we
/// unpack the
/// COM "safe array" into a managed array of bytes.
private int unpackPixels(System.Array dataFromTarget,
    int nPacketSizeBytes,
    byte[] colorPlane, int kk)
{ int N = nPacketSizeBytes >> 2; // divide by 4 b/c there are // 4 pixels per element in safe array
    for (int ii = 0; ii > 8);
        colorPlane[kk+2] = (byte) (fourPixels >> 16);
        colorPlane[kk+3] = (byte) (fourPixels >> 24);
    }
    return kk;
} 

/// Copies color plane pixels into the GDI+ object that is rendered to /// the screen.
private void marshal2GDI(byte [] colorPlane, BitmapData bm, int offset)
{ System.IntPtr scan0 = bm.Scan0;
    unsafe
    {
        byte *p = (byte *)(void *)scan0;
        p += offset;

        int deltaStride = bm.Stride - bm.Width*3;
        for (int ir = 0; ir

-- Video/Imaging DesignLine --