Improving Image Performance by Using Color Lookup Tables

Adobe Developer Support

Technical Note #5121

31 March 1992

Adobe Systems Incorporated

Adobe Developer Technologies
345 Park Avenue
San Jose, CA 95110
http://partners.adobe.com/

PN LPS5121

Copyright © 1991−1992 by Adobe Systems Incorporated. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of the publisher. Any software referred to herein is furnished under license and may only be used or copied in accordance with the terms of such license.

PostScript, the PostScript logo, Display PostScript, and the Adobe logo are trademarks of Adobe Systems Incorporated which may be registered in certain jurisdictions. Other brand or product names are the trademarks or registered trademarks of their respective holders.

This publication and the information herein is furnished AS IS, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies, makes no warranty of any kind (express, implied or statutory) with respect to this publication, and expressly disclaims any and all warranties of merchantability, fitness for particular purposes and noninfringement of third party rights.

Contents

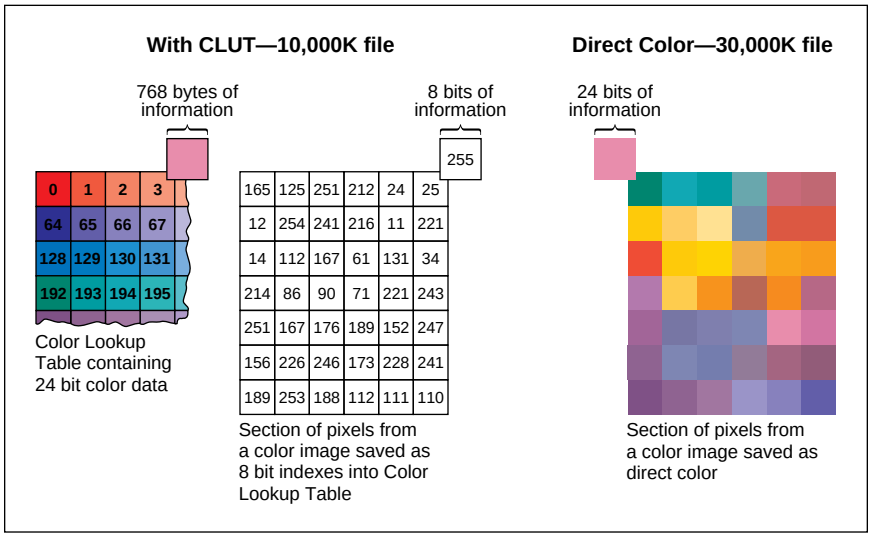
Improving Image Performance by Using Color Lookup Tables

1 Introduction

One of the bottlenecks in processing PostScript™ language files is data transmission time. In PostScript Level 2, one can take advantage of compression filters to help reduce the amount of data transmitted to the printer. Another way to help reduce the size of files, which contain certain kinds of images, is to use the Level 2 indexed color space. The advantage gained by using the indexed color space is not limited to Level 2 devices. It is possible to emulate the indexed color space on Level 1 printers. Both techniques use the same color lookup table (CLUT).

A color table is most effective when a restricted number of colors appear in a particular image. Because the indexes into this table are fewer bytes than the data stored at each location in the table, overall data transmission is reduced, thereby reducing print time.

Figure 1 *8-bit CLUT versus 24-bit RGB image data*



As shown in Figure 1, with an 8-bit CLUT, the data is roughly one-third the size of the 24-bit RGB color image. Each pixel in the CLUT image is represented as an 8-bit index into the CLUT instead of a 24-bit RGB value. Each location in the CLUT stores a 24-bit RGB value.

For instance, if an application has a palette of 256 available colors, the lookup index can be represented by an 8-bit value. The color table is then used to lookup a corresponding 24-bit color value (eight bits of red, green, and blue) on the printer. The alternative is to do the 24-bit expansion on the host and always send down 24-bit RGB data to the printer. For this example, the file size comparison is three to one.

Note that the color lookup strategy does not make sense if your application is printing full 24-bit images. The key to the efficiency of the indexed color approach is limiting the number of colors that can be represented.

This technical note is accompanied by PostScript language code that provides a general strategy for printing images, with and without the use of a color table, and with the optional use of compression filters for printers that support Level 2. The specific files are *clutsetup.ps*, *basicimage.ps*, *colorimage.ps*, and *clutexample.ps*.

The Level 2 indexed color space and the CLUT emulation code have the same basic functionality, but the indexed color space is implemented in the PostScript interpreter itself, whereas the Level 1 CLUT emulation is implemented as a series of PostScript language procedures. This results in faster execution on a PostScript Level 2 interpreter.

The code supplied with this technical note is self-configuring. It is suitable for use on both a Level 1 and Level 2 interpreter and will take advantage of Level 2 functionality if it is available. This allows the application writer to have a single program interface that produces PostScript language code that executes efficiently on both Level 1 and Level 2 devices.

This technical note will show how to use the code provided, discuss the general problems in both the Level 1 and Level 2 cases, and finally will examine the details of how the supplied code is implemented.

2 Using the Code

The code provided with this technical note contains a generic imaging strategy, as well as code to implement a color lookup table for images in Level 1 and Level 2. This section discusses how an application can directly make use of the code for printing color or black and white images.

The CLUT code supplied with this technical note is suitable for RGB images that conform to the following requirements

- the index data is 2-, 4-, or 8-bits (4, 16, or 256 colors)
- the underlying color data is 24-bit RGB (8-bits per component)

The application that generates these images must also be able to generate a lookup table that associates the color indexes with particular colors. This code can be modified to support a variety of conditions.

Note *This code is not suitable for being downloaded outside the server loop as it is provided but may be easily modified for such use. See section 5.1 for details.*

In addition to the CLUT code, code is provided for the following types of images:

- full 24-bit RGB images (including emulation for black and white devices)
- 8-bit grayscale images
- 1-bit monochrome images

As the code is provided, many of the procedure and variable names are fairly long for increased readability. To reduce the size of the prolog and the subsequent scripts produced by your driver, rename the procedures to one or two character identifiers.

If this code is included in a product, you might also want to delete any extra spaces and comments to further reduce the code size. These steps are easily performed with the search and replace features of editors available, but keep a list of the variable names being replaced and to what they correspond for future maintenance of the code.

2.1 Printing Images with CLUTS

There are three PostScript language procedures to call from your application when using this imaging interface. The first is beginimage, and the second depends on the type of data that composes the image. For image data that contains a color lookup table, the procedure is named doclutimage and is one of several conditionally defined procedures. The third is endimage and is called to do general clean-up after the image has been generated.

beginimage

The entry point for the image strategy is the beginimage procedure, which sets up the data acquisition procedure for the **image** operator. Depending on whether the device is Level 1 or Level 2, this can be done in several different ways. If the device is known to support Level 2, the **RunLengthDecode** filter can be used to decrease the size of the image file. (The code can be easily modified to support other compression filters.)

The beginimage procedure expects the following arguments to be placed on the stack in the following order (first is on the bottom).

- image height, image width – The width is the number of samples per scanline, height is the number of scanlines.
- bits per component – The number of bits per sample of the index data in the CLUT case or the number of bits per sample of the image data in the other cases. In the CLUT case this is not the number of bits-per-component of the underlying color space.
- string length – The length of the string used for data acquisition. This should be the length of one scanline of the image for the image code to work properly. See section 5.6 for further discussion.
- size x, size y – The size in the current coordinate system to which to scale the image.
- location x, location y – The location in the current coordinate system to which to translate the image.
- smoothing flag – Boolean indicating whether or not to perform image interpolation on Level 2 devices.
- polarity – This determines how the image data is to be interpreted for the 1-bit cases. If polarity is **false**, 0 bits are painted the foreground color.

- data type – Indicates the format of the image data, which can have the following values:

  0 – Plain binary data

  1 – Plain ASCII hex data

  2 – Run Length encoded binary data

  3 – Run Length and ASCII85 encoded data

The beginimage procedure configures the data acquisition procedure by checking the data type argument passed on the stack.

After beginimage is called, the procedure for imaging the data must be called. This is one of several procedures, depending on the image type being produced. For images using a color lookup table, the imaging procedure is called doclutimage. This will be one of several procedures conditionally defined based on device characteristics such as the number of process colors, whether the device is Level 1 or Level 2 compatible, and so forth. See sections 4 and 5 for a more detailed discussion on how the code decides which procedure to install.

doclutimage

The doclutimage procedure expects two arguments on the stack.

- black and white lookup table – This string contains a black and white version of the lookup table. The conversion from RGB to monochrome should be performed on the host. Each byte represents one 8-bit sample value. This table is used for black and white Level 1 devices, and is always provided to doclutimage. Gray values are 0–255, where 0 is full black and 255 is full white. The length of the table will be (2 ^ bits per component).

- color lookup table – This string contains the color lookup table. The color samples are stored as (RGBRGBRGB...), where each color component is 8-bits, for a total of 24-bits per RGB color sample. Color values are 0–255 (per component) where, for example, 0 is no red and 255 is full red. The length of the table will be (3 × 2 ^ bits per component).

Note *It is important that the lookup tables are the correct length for the code to work properly.*

The index data should be provided directly after the call to doclutimage. If the data is binary, then the indexes must be separated from the word "doclutimage" by exactly one space character. After the imaging procedure executes, the driver should call the endimage procedure, which executes a

restore that undoes the effect of the **settransfer**, **scale**, **translate**, and other operations that occurred as side effects of displaying the image. In addition, any memory that was used by the image procedures will be reclaimed.

See the file *clutexample.ps* for an example of how to use the beginimage procedure with the doclutimage procedure.

Normally, if the PostScript language color operators are used in a page description, the prolog should provide a conditional emulation, because these operators do not exist in all printers. If the **colorimage** operator is used by itself elsewhere in your driver, any conditional emulation should not be named **colorimage** because the Level 1 emulation code checks to see that the **colorimage** is found in **systemdict**. The code could be confused about the type of device on which it is printing.

Having an emulation of the **colorimage** operator could cause poor speed results on Level 1 black and white devices. If such an emulation is needed for some other part of your driver, name the procedure something other than **colorimage** (such as myappimage) and define this name to execute either **colorimage** (on devices where **colorimage** exists) or the emulation procedure (devices without **colorimage**).

2.2 Printing Images without CLUTS

For images that do not use a lookup table, there are several other procedures. These are

- do24image – used for true 24-bit RGB color images
- doimage – used for plain 8-bit grayscale images
- 1bitimage – used for 1-bit images

3 The Level 2 Indexed Color Space

These sections explain how the color lookup table code is implemented.

Color lookup tables are most easily supported on Level 2 devices by using the built-in indexed color space. The indexed color space in Level 2 is always used in conjunction with another color space, such as RGB, CMYK, or one of the CIE based color spaces. The indexed data is expanded through the color table into this underlying color space by the PostScript interpreter. Note that the underlying color space may not be one of the special color spaces: indexed, pattern, or separation.

3.1 Color Spaces

Level 2 implementation of a color lookup table is relatively straightforward. The following is sample code showing the use of the **setcolorspace** operator to set up the indexed color space.

```
/rgbclut (...lookup table...) def
/hival 255 def
[/Indexed DeviceRGB hival rgbclut] setcolorspace
```

The lookup table in this case is represented as a string, where the first byte of a particular color value starts at *index* × *number of color components of color model*. Note that because the color table is a string, it forces the application to put each color component in one byte, or 1/256 quantization. The color table does not have to be represented by a string; it also may be a procedure. Using a procedure is useful for cases in which the sample representation of the underlying color space is a different quantization than 1/256, for example, 1/4096 for 12-bit data.

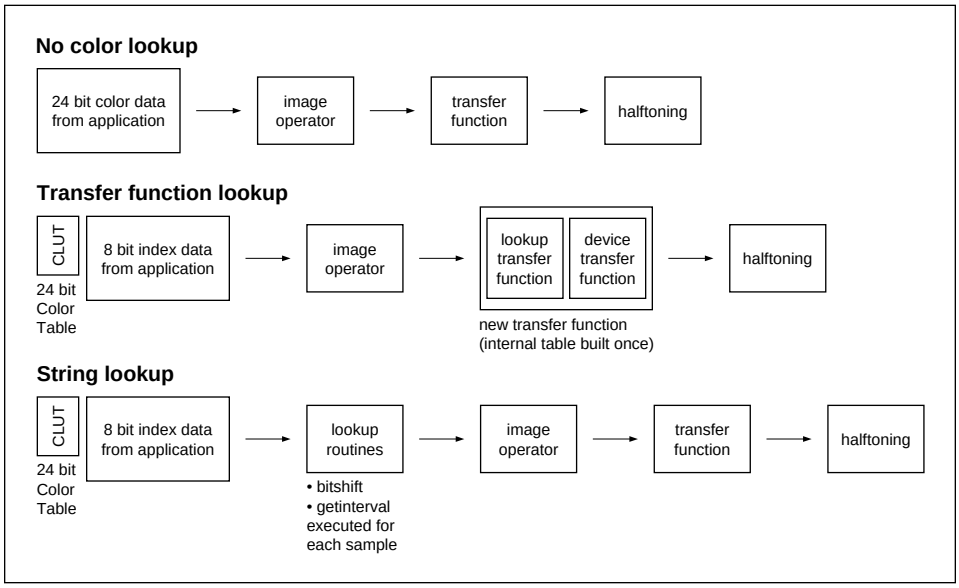
The major difference between implementing a color lookup table in PostScript Level 1 and Level 2 is that in Level 2, the notion of the current color space also affects the **setcolor** operator. One implication is that the parameters passed to the **setcolor** operator may also make use of the color lookup table. This generally will not save nearly as much transmission time as when printing images. If you want to make use of the **setcolor** operator with an indexed color space in your driver, emulate the **setcolor** operator and indexed color space functionality for Level 1 compatibility. The code supplied with this technical note is oriented toward images and does not perform this emulation.

4 Emulating Indexed Color Space in Level 1

Creating a fast Level 1 emulation of an indexed color space is significantly more complex than using the Level 2 built-in implementation. The following section discusses the important issues involved in creating a suitable emulation.

Because PostScript Level 1 supports only the gray, RGB, HSB, and CMYK color models, you are restricted to these models for the base color space of your data when using the emulation code. The CIE based color spaces are not supported in PostScript Level 1.

Figure 2



There are essentially two ways to emulate an indexed color space in PostScript Level 1. The first method involves using the transfer function and is very fast but requires certain knowledge of the output device to produce correct results. The second method involves string manipulation and is much slower but is completely device-independent. Both methods are provided in the emulation code and are conditionally used in different cases.

As shown in Figure 2, string lookups are slower than using the transfer function as a lookup table because several PostScript language operators must be executed for each sample value. Since the data is still much smaller than full 24-bit data, the string lookup strategy is still faster over slow communication lines.

4.1 Using the Transfer Function

The transfer function is a part of the graphics state and is defined with the **settransfer** operator that is generally used to compensate for non-linear response of an imaging engine. Since the transfer function is a procedure, it can be used in very powerful ways. In our case, we will use the transfer function to do our color lookup. If you are unfamiliar with the transfer function, refer to section 6.3 of the *PostScript Language Reference Manual, Second Edition*, for more information. In general, a clear understanding of section 6 of the manual is very helpful for understanding indexed color spaces and the CLUT emulation code.

Different output devices have different numbers of process colors they can use directly to generate colors. For instance, many screen displays are driven by RGB signals. Most color printers generate CMYK or CMY output; monochrome printers generate only one color. To use the transfer function machinery as our lookup table, we need to know the number of process colors the current device is generating.

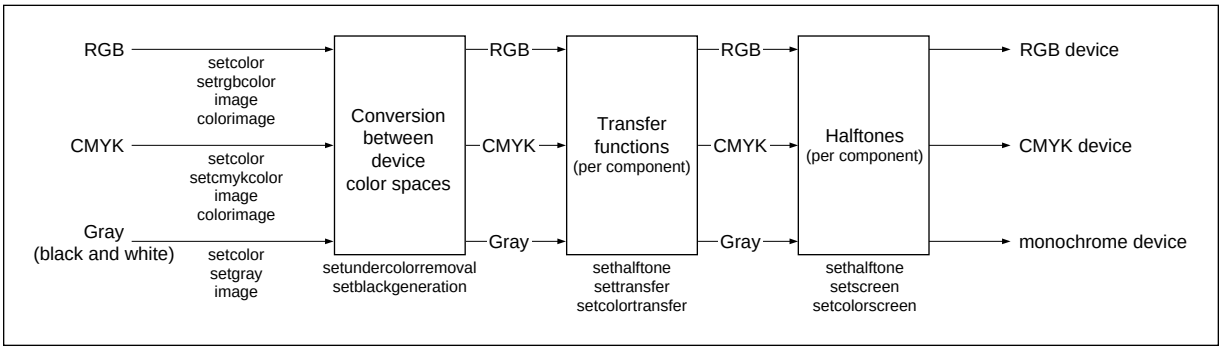
If we know the number of process colors that the output device is set to produce, we can gain speed in the lookup table by having the PostScript interpreter do the lookup through the transfer functions. Thus, using the transfer function is really three separate cases: one-color (black and white), three-color (RGB or CMY), or four-color (CMYK), depending on the device.

From a programming point of view, the transfer function expects a number on the stack from 0 to 1 that represents a gray value, and it leaves another value on the stack that represents a gray value with any device irregularity compensation performed. An analogous operator called **setcolortransfer** exists in Level 1 interpreters that support color output.

A color transfer function is actually four transfer functions in one. Each procedure corresponds to a color component R, G, B, and Gray. It is important to note that transfer functions are always specified in additive (R, G, B, Gray) color space. This is explained more fully in section 4.4.

When the transfer function is set up with the **settransfer** or **setcolortransfer** operators, the PostScript interpreter builds an internal lookup table from the transfer function by calling it repeatedly with values between zero and one. This table is used as the final stage transformation of the input color into device color before halftoning. Because this lookup table is built once and used during the time that the transfer function is in effect, the performance is very good. The transfer function procedure is executed only the number of times necessary to build the transfer lookup table.

Figure 3 *Device color spaces*



As shown in Figure 3, using the transfer function as a lookup table depends on knowing the number of process colors of the output device so that we can specify input data in the same color space as the output device. If data is specified in a different color space, the data may undergo some conversion resulting in incorrect indexes passed to the transfer function.

The key to using the transfer function as a lookup table is to avoid any color-space conversion between the output of the **image** operator (the index data) and the input to the transfer function (our lookup table), so the indexes are not modified in any way (see Figure 3). This is accomplished by specifying the index data to the **image** or **colorimage** operator in the same color space as the color space of the output device.

For example, for a device generating RGB process colors, we must specify the indexes using the RGB form of the **colorimage** operator so that no translation occurs between the index data (**colorimage**) and the lookup table (the transfer function). Thus, there is a separate piece of PostScript language code to perform the color table lookup for each color model. Each piece of code uses the color table to specify the index data in terms of the devices' output color space.

The theory of operation of our lookup table implemented as a transfer function is as follows.

1. Multiply the value on the stack (between 0 and 1) by the number of entries in the lookup table to produce an index.

2. Extract the color value at the index.

3. Divide by 255 to produce a number between 0 and 1, which represents the color value.

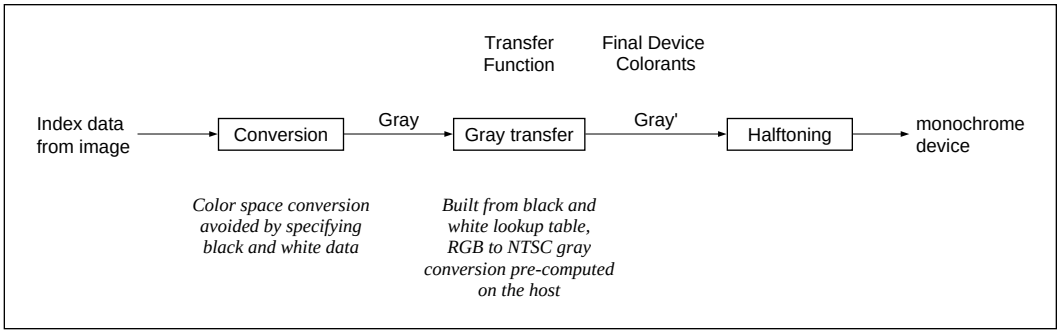
It is important to note that the color lookup table function we create to pass to the transfer function does not replace the current transfer function. Instead, the two transfer functions are concatenated. This important step must happen in the correct order.

The normal transfer function is used to correct non-linear response in the output device. If it is concatenated before the color lookup function, the results will be incorrect since the index values will be modified. Instead, the color lookup function must occur first, followed by the original transfer function as shown in Figure 2.

4.2 Black and White in Theory

The theory of the black and white case is the easiest to understand. The transfer function is used to build a lookup table using a black and white lookup table.

Figure 4 *One-color output device*



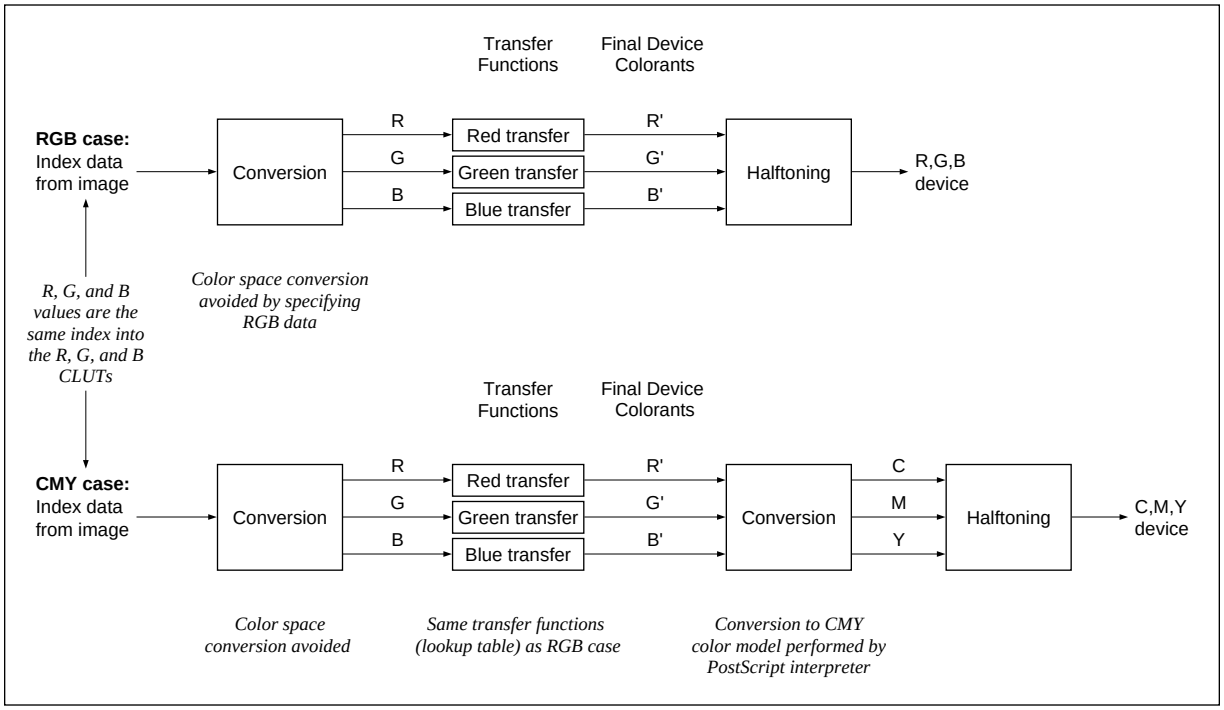
On a black and white device, the lookup table is a single transfer function.

The emulation code provided uses a black and white lookup table in addition to an RGB lookup table. The black and white table is ignored in Level 2, and all Level 1 cases except the one-color output device. It is possible to create a black and white lookup table from the RGB CLUT by executing PostScript language code, but it is faster to do this conversion on the host and send down a pre-computed black and white lookup table in addition to the RGB table. Note that both tables are always sent. The application does not need to know which case is being executed.

4.3 Three-Color in Theory

If the output device that we are printing to is generating three process colors, then we must use the RGB form of the **colorimage** operator to specify the indexes to the transfer functions for this device.

Figure 5 *Three-color output device*



Three-color devices require three transfer functions. Because the transfer functions are always specified as an additive color model, the same lookup table is valid for both RGB and CMY output devices.

Because we want to pass the same color value index to each component of the transfer function, the index data read by the data acquisition procedure is **dup**'d on the stack twice. This provides three identical strings to the **colorimage** operator, which is expecting three strings: one for the red component, one for green, and one for blue.

During the execution of **colorimage**

1. The PostScript interpreter passes identical sets of data to the transfer functions.

2. The transfer functions extract the appropriate component from the RGB CLUT at the same index.

3. The color samples are stored as (RGBRGBRGB...), where each color component is 8-bits. That is, the red transfer function extracts the red component at (index × 3), the green transfer function extracts the green data at ((index × 3) + 1), and so forth.

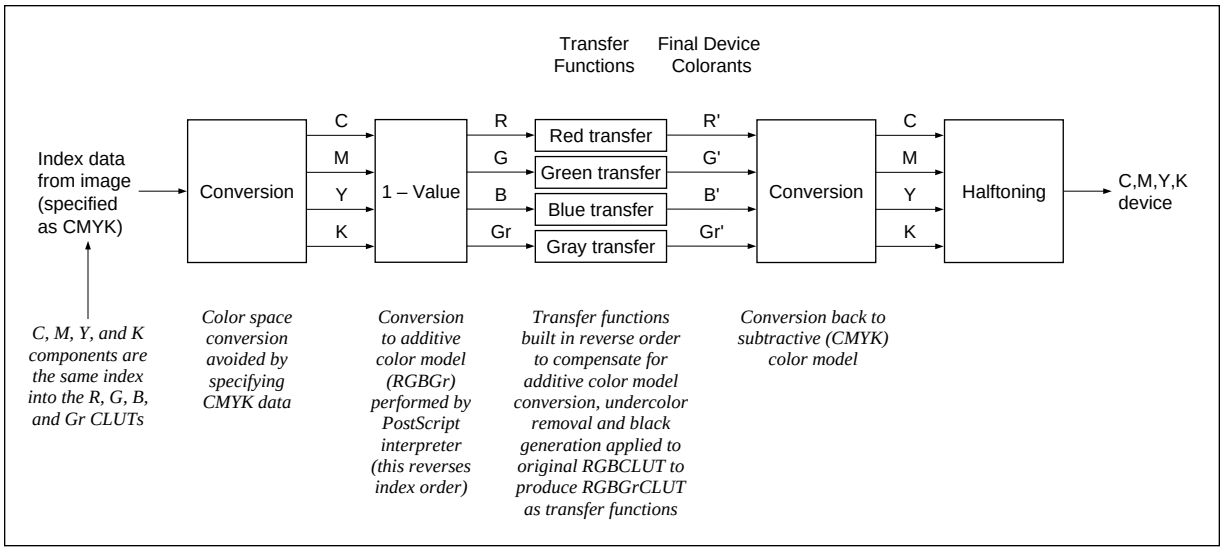
4. The data produced by the transfer functions is passed on for halftoning. In the CMY output case, the data is first converted back from the RGB color values to the CMY color values.

Note there are always four components passed to the **setcolortransfer** operator. In this case, since we are printing on a three-color device, the fourth (gray) component passed to **setcolortransfer** is not used when the input data is specified as RGB data. (The gray transfer function is used for a three-color device in two cases: When printing to a device with one process color, such as a color printer with a black ribbon installed, or when printing black and white data specified with the **image** or **setgray** operators on a three-color device. Neither of these cases occurs in this situation.)

4.4 Four-Color in Theory

Printing to a device generating four process colors is the most complex case because the data goes through a number of transformations, and because the black component of the image must be generated from the RGB data. To produce the correct results on a four-color device, we must run the RGB data through the printer's built-in black generation and undercolor removal functions to produce a fourth component for the black (K) process color.

Figure 6 *Four-color output device*



The RGB table is not used directly in the four-color case. A CMYK table is built from the RGB data by using the PostScript interpreter's built in RGB to CMYK conversion process. This is accomplished in PostScript by using the sequence **setrgbcolor currentcmykcolor**.

Because we want to pass the same color value index to each component of the transfer function, the index data read by the data acquisition procedure is **dup**'d on the stack three times. This creates four identical strings to pass to the **colorimage** operator, which is expecting CMYK data. This is important: Although the data is really index data into our color table, the PostScript interpreter sees it as true CMYK image data. This has implications when the data gets to the transfer function stage since the operand and result of a transfer function are always specified as if the component were additive (red, green, blue, or gray). That is, larger numbers indicate lighter colors.

If the component is subtractive (cyan, magenta, yellow, black, or a separation), the PostScript interpreter converts it to additive form by subtracting it from 1.0 before passing it to the transfer function. Because of this, the transfer functions must perform their lookup in the reverse order, so the CMYK CLUT used by the transfer function is actually built in reverse. That is, the color table entry corresponding to a value of 0 in the input sample data is the last entry in the CMYK lookup table.

Since the output device has four process colors instead of three, we must create a fourth lookup table for the black component of the image from the RGB data. The process of converting RGB data to CMYK data is fairly straightforward. We can simply use **setrgbcolor currentcmykcolor**, and rely on the interpreter's current functions for black generation and undercolor removal. (See section 6.2 of the *PostScript Language Reference Manual, Second Edition* for a detailed explanation of how this is accomplished). In the sample code, a new lookup table is created that contains the CMYK data.

However, the **currentcmykcolor** operator will return color values that are in a subtractive (CMYK) color model, and our transfer functions must be specified as additive color model (RGBGray). To eliminate this problem, the values are subtracted from 1.0 before being added to the new lookup table.

4.5 Using String Lookups

Although using the transfer function is the fastest method of implementing the color table lookup, there are several cases where it cannot be used because information about the number of output colors is not available or we are generating color separations. In these cases, we must use string lookups.

The primary difference between using string lookups and using the transfer function is where data lookup actually occurs in the chain of the data stream (see Figure 2). In the transfer function case, the color table lookup happens after the data is passed to the **image** (or **colorimage**) operator. In the string lookup case, the lookup happens in the data acquisition procedure called by the **image** operator, so the data has already been expanded before it gets to the **image** operator.

This means that the PostScript interpreter is doing more work in terms of executing procedures, and this in turn means that this method is much slower than using the transfer function method. Fortunately, using this method is not common. This case can still provide an overall improvement in print speed for very slow communication lines, such as 9600 baud serial.

As each byte is read from the input stream, it is used by the data acquisition procedure passed to the **colorimage** operator as an index into the color lookup table. The byte is expanded by using the **getinterval** operator to retrieve a series of data bytes from the lookup table.

The key to operation in this case is that the lookup table is implemented as part of the data acquisition procedure for the **colorimage** operator.

5 Code Walkthrough

This section includes a walkthrough of the code and a general discussion of other techniques revealed in the code that apply more generally to PostScript language programming.

5.1 Self-Configuration

As with much of the code currently provided by Adobe Developer Support, the CLUT code is self-configuring; it will load into memory only the procedures that are needed for a particular printer, while maintaining a consistent interface to the application.

The key to the self-configuring code is the use of the **save** and **restore** operators. Consider the following code:

```
skipme? { /dontloadme save def } if
...
... conditionally defined code
...
skipme? { dontloadme restore } if
```

The Boolean skipme? is set earlier in the code depending on whether or not the following bit of code should be loaded. If skipme? is true, the first condition above will execute, causing a **save** to occur before the definition.

After the code is loaded, skipme? is checked again, and if true, the **restore** sets the state of memory back to what it was before the conditional code was defined. This undoes the effect of any definitions that happened between the two conditions. It is important to note that for this code to work properly, the code between the two conditional checks must not alter the value of skipme?. Also note that the conditionally defined code must not leave any composite objects on the stack that would cause an **invalidrestore**.

Code similar to that above is used for two purposes in the files *basicimage.ps* and *colorimage.ps*: to load definitions that are appropriate for Level 1 versus Level 2, and to choose between code, in the Level 1 case, based on the number of process colors that the current output device is printing. This decision process is explained in the following sections.

Note that because this code only **def**'s the case it needs to execute properly when it is sent to the printer, it is not suitable for being downloaded outside the server loop in printers that have a variable number of process colors.

For example, if this code is downloaded outside the server loop to a printer that is set for four-color (CMYK) output and afterwards the output is switched to one color (black and white), the code will produce incorrect results. If your application needs to use this code outside the server loop, you should do several things.

- Define each of the different doclutimage cases as separate names, and define a doclutimage procedure in the document setup section of your output that checks the current state of the output device and calls the correct procedure based on this information. This is not the default because it uses much more memory, since all the separate doclutimage cases must be loaded into memory all the time.

- Remove the **save**/**restore** conditional definition code surrounding each of the doclutimage procedures.

- Check the code to make sure that there are no variable name conflicts between the doclutimage cases.

5.2 Determining the Number of Colors

The following sections are a walkthrough of the code provided for emulation of the indexed color space for images. It will be helpful when reading this section to be able to refer to the actual code, preferably with a text editor that has a search function, but a printout will suffice. Most of the work happens in the files *basicimage.ps* and *colorimage.ps*.

If the device supports PostScript Level 2, the code that uses the Level 2 indexed color space is installed.

If the device is Level 1, the code installed depends on the output characteristics of the device. There are four distinct cases: the single-color case, the three-color case, the four-color case, and the conservative case. We must know the number of output colors to use the fast transfer function lookup code, otherwise we resort to the conservative case code. See section 4.1 for a complete discussion.

The code defines a variable called ncolors that represents the number of process colors the current device is using to output. The ncolors variable is used to conditionally define sections of code that are appropriate for the current device by using the techniques discussed in section 5.1.

To determine the number of colors of the current output device, the code first verifies whether the operator called **colorimage** exists. If not, we know this must be a black and white Level 1 printer since the color extensions are present in products capable of producing three or four process color output. At this point, ncolors is set to 1.

If the **colorimage** operator does exist, the code then looks in **statusdict** for an entry called **processcolors**. The **processcolors** operator is a device-dependent operator that exists on some Level 1 color printers. It returns the current number of colors the device is using to generate output. If **processcolors** exists, its value is used as the new value for ncolors.

If **processcolors** does not exist, the code checks for the existence of the **deviceinfo** operator. The **deviceinfo** dictionary, present on some Display PostScript™ systems contains, among other things, a key called **Colors**, which represents the current number of process colors. If this key is found, ncolors is set to the value which it contains.

If either **processcolors** or the **deviceinfo** dictionary was found, the code then verifies that the **colorimage**, **setcolortransfer**, **currentcolortransfer**, and **currentcmykcolor** operators are found in **systemdict** by the normal dictionary lookup mechanism. That is, we want to make sure that we are going through the “real” transfer function machinery.

Note that we cannot simply check whether the operators exist in **systemdict**. We must check that the operators in **systemdict** will be found through the dictionary lookup mechanism. Possible situations in which these operators would not be found in **systemdict** include generating color separations and cases in which an emulation package for the color operators has been downloaded. If neither **processcolors** nor the **deviceinfo** dictionary is available, the value of ncolors is set to 0, which will cause the conservative case definitions to be loaded.

Finally, when doclutimage is called, one of the following four cases will be executed to render the bitmap and perform the color lookup. Each case has an identical interface. It does not matter to the application which procedure is actually called. The doclutimage procedure expects a black-and-white and a color lookup table on the stack in the form of strings. The various flavors of doclutimage are described in the following sections.

After the imaging procedure executes, the driver should call the endimage procedure, which executes a **restore** that undoes the effect of **settransfer**, **scale**, **translate**, and other operations that occur as side effects of displaying the image. In addition, any memory used by the image procedures will be reclaimed.

5.3 Single Color (Black and White) Case

The easiest device to create a lookup table for is the single-color output device. Rather than using the color lookup table, a black and white version of the table is created on the host and sent down with the color table. Both tables are passed in on the stack to the doclutimage procedure, which **def**s the black and white lookup table as bwclut, and pops off the color lookup table since it will not be used in this case.

The next step in the black and white doclutimage is to determine the expansion factor for the data, that is, the amount that a 0–1 gray value must be multiplied by to produce an index into an n-element lookup table. For example, for an 8-bit lookup table, the expandfactor will be 255. Note that the value is 1 less than the number of elements in the lookup table, because the first element in the table is element 0.

After the expandfactor is determined, the lookup table transfer function is built by concatenating the procedure expandbw with the current transfer function. The expandbw procedure actually performs the lookup into the table. Remember, because the transfer function builds an internal table, the expandbw procedure will be called only when **settransfer** is executed, not for each sample value in the image.

Let’s examine the expandbw function.

```
/expandbw
{
   expandfactor mul round cvi bwclut exch get 255 div
} bind def
```

A transfer function procedure is passed a number on the stack from 0–1 and returns another number in that range. In this case, we must convert the 0–1 range into a suitable index into our color table, so the expandfactor **mul round cvi** does this. We multiply by the expandfactor and convert the result into an integer, which is the only suitable type to use with **get** and the bwclut string.

After we convert to an index, we **get** the value stored at that location in the lookup table and divide it by 255 to return to the 0–1 range expected by the transfer function.

Finally, the actual image procedure is set up and executed. The setupimageproc defined by beginimage is used as the data acquisition procedure for the image operator, and the **iw**, **ih**, and **bpc** values that were defined in the call to beginimage are now passed to the **image** operator.

5.4 Three-Color Case

The three-color case is somewhat more complicated than the one-color case, though the basic theory of operation is the same. Note in the code that there are some definitions that are shared between the three-color and four-color cases.

The first step in the three-color doclutimage procedure is to define rgbclut, and pop off the black and white lookup table since it will not be used for this case.

The next step is to define a separate lookup table for each of the color components red, green, and blue. This is accomplished in setuprgbcluts, which is shared between the three- and four-color cases. The setuprgbcluts procedure first defines the variable bit3x. Its use is similar to the variable expandfactor in the black and white case. It is used to calculate the index position in the color table for the current value. The value of bit3x is (length of rgbclut - 3). Note that setuprgbcluts also defines a variable called bit1x, which is the number of elements in the rgbclut. The bit1x is not used in the three-color case, but is used in the four-color case.

The setuprgbcluts procedure defines the rclut, gclut, and bclut by calling the defsubclut procedure. By careful use of the **getinterval** operator, the defsubclut defines these as three separate lookup tables. They are, in fact, sub-strings of the same lookup table (rgbclut). This is done to make indexing into the tables faster.

The next step in doclutimage is to use the spconcattransfer procedure to concatenate the existing transfer functions for the red, green, blue, and gray components to the lookup table functions. This is analogous to what happened for the black and white case where the transfer functions were concatenated and passed to **settransfer**. In this case there are four transfer functions (one for each color component—red, green, blue, and gray) passed to **setcolortransfer**.

As discussed in section 4.3, the gray procedure is always ignored in the three-color/three-color device case. Because of this we can simply **dup** the blue transfer function to create the fourth function passed to **setcolortransfer**.

Each of the lookup functions in spconcattransfer has the form xclut ncompute. The ncompute is actually the lookup function and is passed in on the stack to the spconcattransfer procedure. In this case, it is the procedure 3compute, which is analogous to the black and white procedure expandbw. This procedure

1. expects a number on the stack in the range 0–1,

2. converts the number to an index value into the appropriate CLUT (this will be rclut, gclut, or bclut, depending on which transfer function component is being called),

3. and performs a **get** out of the CLUT, and converts that value back into a color value in the range 0–1.

5.5 Four-Color Case

The four-color case is the most complex, since there are four transfer functions, and the RGB data is being converted to CMYK data, which means that the generation of a black component is involved. This causes several transformations to the data that must be carefully examined for a complete understanding of the code.

The first step in the four-color case of the doclutimage procedure is to define the rgbclut and pop the black and white clut off the stack.

Next, doclutimage checks to see if the current CMYK color table needs to be recomputed. Because this step is time-consuming, it should be done only when necessary. If there are several images in the same document which

share a common color table, the driver could be smart enough to avoid rebuilding the CMYK table from the RGB data between each one. This is accomplished through the invalidcolortable? variable.

Setting invalidcolortable? to true will cause the CMYK table to be recomputed. This value must be true for the first execution of the doclutimage procedure, unless computecmykclut has explicitly been called previously in the job. Changes to the black generation and undercolor removal functions will invalidate the color table, as will a new rgbclut.

Note *The driver should always set invalidcolortable? correctly no matter which color case the job is executing. The other cases simply ignore this variable. At worst, it can always be set to true, although this might cause unnecessary computations to occur.*

Assuming that invalidcolortable? is true, and the cmykclut has not yet been computed, computecmykclut will be called. computecmykclut first calls the procedure setuprgbcluts to create rclut, gclut and bclut (see section 5.4). Next, it defines the variable bit4x, which is the value of the last index into the cmyk lookup table. The bit4x variable is analogous to the bit3x variable used in the three-color case. It is also used to convert the 0–1 data from the transfer function to an index into the CMYK table.

The next step in computecmykclut is to actually define the cmykclut and the four sub-cluts, which, like rclut, gclut, and bclut, are indexes into the main cmykclut. Finally, the actual business of filling up the cmykclut array begins. This is accomplished by using a **for** loop to extract the RGB triplet from the rgbclut for each possible index value. The RGB data is then converted into CMYK data by executing **setrgbcolor currentcmykcolor**. The resulting CMYK data is stored into the cmykclut.

Before each element in the lookup table can be passed to the **setrgbcolor** operator, however, it must be divided by 255 to convert from the lookup table's representation (0–255) to the 0–1 range which **setrgbcolor** operator expects. The **currentcmykcolor** then returns four integers in the 0–1 range, but before these values are multiplied by 255 to convert them into 8-bit quantities suitable for the lookup table, they are subtracted from 1.0 to convert the subtractive color model returned by **currentcmykcolor** to the additive color model. This is done because the transfer functions are always specified as additive (see section 4.4).

Finally, stuffclut stores these values in the new cmykclut in reverse order because the data passed to **colorimage** in the four-color case is subtracted from one before being passed to the transfer functions.

The data acquisition procedure built for the four-color case duplicates the string value returned by the setupimageproc procedure three times to create four strings of data to be sent to the **colorimage** operator, which is expecting CMYK data. Finally, **colorimage** is called.

5.6 Conservative Case–Unknown Number of Colors

The conservative case of color lookup is the most intuitive method of creating lookup table code. However, because it may involve bit shifting operations for index data, and because several PostScript operators are executed for each sample value, it is much slower. Fortunately, this is not the common case.

As with the three- and four-color cases, the first thing to happen in the conservative doclutimage procedure is to **def** the rgbclut and pop off the black and white lookup table. Next, createxpandstr is called to create a string long enough to hold the expanded data from the original data acquisition procedure. That is, the image data passed to the **colorimage** operator is significantly larger than the index data read by the data acquisition procedure (setupimageproc), so we must allocate a new string to hold this expanded data.

The size of the expanded string is the size of the original data acquisition procedure's string times the amount the data expands. The amount to expand the data is based on the size of the input data to the **colorimage** operator. For instance, if the depth of the index data is 8-bits, and we need 24-bit data from the color table, then the expansion factor is 3 (24 / 8 = 3). If the depth of the index data is 2-bits, and we want 24-bit color data, then the expansion factor is 12, and so on.

At the same time expandstring is created, the proper lookup function is defined, since this is also a function of the bits per component of the index data. This lookup function is defined as mylookup, and is one of 8lookup, 4lookup, or 2lookup.

Finally, the original data acquisition procedure is concatenated with the lookup function to create a new data acquisition procedure to pass to the **colorimage** operator.

Appendix A: Variable Cross Reference for the Supplied Code

The definitions in this section are listed by file, then in order of appearance within the given file (roughly).

1 Definitions Needed Prior to Execution

Refer to the file *CLUT setup.ps* for these definitions, which are required to execute the example code in *CLUT example.ps.*

- level2 –(boolean) A Boolean indicating whether or not this printer supports PostScript Level 2.
- bd – (procedure) A convenience procedure for executing **bind def**.
- xd – (procedure) A convenience procedure for executing **exch def**.
- ld – (procedure) A convenience procedure for executing **load def**.
- Invalidcolortable? – (boolean) Must be set to true if CLUT becomes invalid. This is used to determine whether to recompute the cmykclut in the four process color case. This is important if there are several images in the same document that have lookup tables that are somehow different. This must always be set to true for the first image so that the cmykclut is computed at least once.
- myappcolorspace – (name) The underlying color space for Level 2 devices that use the indexed color space. This code defaults to **DeviceRGB** as the underlying colorspace, but can be overridden by defining myappcolor-space to be some other color space within the document setup or script of the job.

2 Basicimages_level12.ps

- dontloadlevel1 – (boolean) Don't load any of the Level 1 cases if this is set to true.
- dontloadlevel2 – (boolean) Don't load any of the Level 2 cases if this is set to true.
- polarity – (integer) This determines how the data in a 1-bit image is to be interpreted, that is, whether a color value of 0 represents the foreground or background color.
- smoothflag – (boolean) This is a Boolean to turn on Level 2 bit smoothing code for 1-bit images. This has no effect on a Level 1 printer.
- mystring – (string) This string is used by the data acquisition procedure to store image data read from currentfile. It should be the length of one scanline of the image.
- iw – (integer) The width of the current image in samples; the number of samples per scan line.
- ih – (integer) The height of the current image; the number of scan lines.
- bpc – (integer) Bits per component of the image. Note that this is the number of bits per index value in the CLUT case or number of bits per component of the image data in all other cases.
- beginimage – (procedure) General image setup procedure that provides several functions including determining the type of data to be expected, scaling and translating the coordinate system, and generating a **save** level around the following image. beginimage defines the following variables and procedures.

  im_save
  setupimageproc
  polarity
  smoothflag
  mystring
  bpc
  ih
  iw

- im_save – (save object) This is used in beginimage to wrap the entire image in a **save**/**restore**.
- endimage – (procedure) This does a **restore** to im_save.

- myimagedict – (dictionary) This is used in the Level 2 case to set up the **image** operator.

- setupimageproc – (procedure) This sets up the procedure used for data acquisition. setupimageproc is defined as one of the following procedures during beginimage depending on the data type passed to beginimage on the stack:

  setup2binaryproc – For use with run length encoded data

  setup2asciiproc – For use with run length and ASCII base-85 encoded data

  setup1binaryproc – For use with plain binary data

  setup1asciiproc – For use with plain hex data

2.1 Images without CLUT

- 1bitimage – (procedure) This procedure is for 1-bit black and white images.

3 Colorimage_level12.ps

The definitions below are defined in *colorimage_Level12.ps*

- doclutimage – (procedure) This procedure is called to do an image with a color lookup table. It is conditionally defined as one of several procedures that handle the several different cases for doing CLUT images (see sections 2 and 3 for a complete discussion). There are four cases for Level 1 and one case for Level 2. Cases for Level 1 are

  one-color (black and white) case

  three-color (RGB or CMY) case

  four-color (CMYK) case

  Conservative case

- startnoload – (procedure) This procedure is used in self-configuring code to start ignoring definitions.

- endnoload – (procedure) This procedure is used to stop ignoring definitions and restore memory to the state that was saved at the time startnoload was executed.

- noload – (save object) This is the save level variable for above.

- testsystemdict – (procedure) This procedure will see whether a key will be found in *systemdict* when using the dictionary lookup mechanism.

- ncolors – (integer) Represents the number process colors that the device is generating. This is only used for the Level 1 CLUT code. If ncolors is 0, the conservative case definition of doclutimage is used. Possible values are: 0, 1, 3, 4.

- expandbw – (procedure) This procedure is the black and white case lookup function concatenated with the existing transfer function for black and white images.

- bwclut – (string) This string is the lookup table for black and white (one-color) case.

- expandfactor – (integer) 8 bits = 255, 4 bits = 15, 2 bits = 3 This is the number by which to multiply the index data from the **image** operator to produce an index into the bwclut.

- concatutil – (procedure) This procedure is used to concatenate two procedures. Called by spconcattransfer.

- spconcattransfer – (procedure) This procedure is concatenates existing color transfer functions with color table lookup functions.

- defsubclut – (procedure) This procedure helps define *sub-cluts*. Called by setuprgbcluts, computecmykclut.

- ncompute – (procedure) This procedure is a compute function passed in to spconcattransfer.

- setuprgbcluts – (procedure) This procedure is creates sub-cluts from rgbclut.

- computecmykclut – (procedure) This procedure is builds the cmykclut from the rgbclut by using the PostScript interpreter's black generation and undercolor removal functions.

- ftoint – (procedure) This procedure is converts 0–1 value (trans func) into 0–255 value for indexing into CLUT

- 3compute – (procedure) This procedure is computes a color value in the rgbclut given an index.

- 4compute – (procedure) This procedure is computes a color value in the cmykclut given an index.

- bit3x – (inegert) This is the last index in RGB table (actually, length - 3)

- bit4x – (integer) This is the last index in CMYK table (length - 4)
- rgbclut – (string) The real RGB CLUT from which the following substrings are defined: rclut, gclut, and bclut
- cmykclut – (string) The real CMYK CLUT from which the following substrings are defined: cclut, mclut, yclut, kclut.

3.1 Conservative Case Definitions

- mylookup – (procedure) This procedure is is defined to one of the following in the doclutimage proc:

  8lookup –lookup for 8-bit data

  4lookup –lookup for 4-bit data

  2lookup –lookup for 2-bit data

- lookupandstore – (procedure) This procedure is called by the mylookup procedure to put data into the expanded data string (mystringexp).
- colorexpand – (procedure) This procedure is looks up data from rgbclut and puts it in mystringexp by calling mylookup.
- createexpandstr – (procedure) This procedure is creates the string called mystringexp that is ($m \times$ length(mystring)), where m is the multiplier value for the expanded data string. Insure that the size of the expand string does not exceed the maximum string length limit.
- mystringexp – (string) String in which the expanded data from mystring is stored.
- mystring – (string) The original data acquisition procedure's string.

3.2 Images without CLUT

- rgbtogray – (procedure) This procedure is used by do24image to emulate the three-color **colorimage** case on black and white devices without a built-in colorimage.
- do24image – (procedure) Imaging procedure used for a true 24–bit RGB images.
- doimage – (procedure) Imaging procedure used for 8-bit grayscale images.

Appendix B: Changes Since Earlier Versions

Changes since August 12, 1991 version

- Document was reformatted in the new document layout and minor editorial changes were made.

Index

A, B

C

D

E

F

G, H

I, J, K

L

M

N, O

P, Q

R

S

T, U, V, W

X, Y, Z