Adobe Technical Journal

Color Services and the Color Picker Plug-in

Rev. 2

Andrew Coven
Photoshop Developer Support Engineer
Adobe Systems, Incorporated

gapdevsup**@adobe.com**

Introduction

Color Services and the Photoshop 4.0 Color Picker plug-in

The Adobe® Photoshop® 4.0 application programming interface introduces a new plug-in type: *Color Picker* plug-ins. Color Picker plug-ins allow you to develop and implement your own style of custom color picking mechanism, with the goal to provide similar results and functions to the Photoshop or OS Color Pickers. While the new Color Picker plug-in architecture is very straightforward, the Color Services suite, a long-time complex and inconsistent architecture, has been a bit of a mystery. This article will detail the Color Services suite of functions and structures, and discuss the mechanics of the Color Picker plug-in module.

Scope of this article

Refer to the SDK

Intimate details on all the parameters and callback suites are available in the Adobe® Photoshop® 4.0.1 SDK, which is available at Adobe's web site:

`http://www.adobe.com/supportservice/devrelations/sdks.html`

This article will only address the callbacks and structures that are pertinent to the Color Services suite and the Color Picker plug-in module. There is much more than is covered in this document. Read the SDK for more detail.

Macintosh or Windows?

Color Services, chosing a color, converting between color spaces, and Color Picker plug-in modules are all part of the Adobe Photoshop API. This means that, except in a few rare exceptions and in the specific cases of opening dialog windows and other platform resources, the callbacks, data structures, and parameters are all exactly the same on both Macintosh and Windows. This article shows Macintosh user interface examples, but the discussion and examples are comparable, if not exactly the same, on Windows.

Color Services Suite

The Color Services suite is a set of functions that are specific to working with color data. The suite is used to perform any of four operations:

1. Return a sample point.
2. Return either the foreground or background color.
3. Choose a color using the user's preferred Color Picker (which, in Photoshop 4.0, can be a custom Color Picker via a plug-in).
4. Convert a color from one space to another.

The next sections of this article will discuss each operation in detail.

Returning a sample point

Returning the current sample point is one of the more straightforward of the Color Suite's functionality. All basic Color Suite functions are accessed and triggered by handing a filled-out `ColorServicesInfo` struct to the Color Services callback. The `ColorServicesInfo` struct is defined as:

Figure 1: ColorServicesInfo struct

```
typedef struct ColorServicesInfo
{
   int32 infoSize;                  // Size of ColorServicesInfo record.
   int16 selector;                  // Operation.
   int16 sourceSpace;               // Color space of input color.
   int16 resultSpace;               // Color space of output color.
   Boolean resultGamutInfoValid;    // Has resultInGamut field been set?
   Boolean resultInGamut;           // Is returned color in gamut for printing?
   void *reservedSourceSpaceInfo;   // Must be NULL.
   void *reservedResultSpaceInfo;   // Must be NULL.
   int16 colorComponents[4];        // Actual components of input or output color.
   void *reserved;                  // Must be NULL.
   union
   {
      Str255 *pickerPrompt;         // Picker prompt string.
      Point *globalSamplePoint;     // Where to sample.
      int32 specialColorID;         // ID of which color to return.
   } selectorParameter;
}
```

You can fill this struct out yourself, however there is a routine to initialize a `ColorServicesInfo` struct to typical values. The routine is in the *PIUtilities* library, which is the `PIUtilities.c` and `PIUtilities.h` files in the Photoshop SDK's `Examples/Common` folder:

Figure 2: CSInitInfo routine

```
OSErr CSInitInfo (ColorServicesInfo *ioCSinfo)
{
   OSErr err = noErr; // Assume no error, initially.

   if (ioCSinfo != NULL)
   {

      // Zero color components:
      CSSetColor(ioCSinfo->colorComponents, 0, 0, 0, 0);

      // Selector is one of these:
      // plugIncolorServicesChooseColor,
      // plugIncolorServicesConvertColor,
      // plugIncolorServicesSamplePoint,
      // plugIncolorServicesGetSpecialColor:
      ioCSinfo->selector = plugIncolorServicesConvertColor;

      // sourceSpace and resultSpace can be:
      // plugIncolorServicesRGBSpace,
      // plugIncolorServicesHSBSpace,
      // plugIncolorServicesCMYKSpace,
      // plugIncolorServicesLabSpace,
      // plugIncolorServicesGraySpace,
      // plugIncolorServicesHSLSpace,
      // plugIncolorServicesXYZSpace
      // resultSpace could also be: plugInColorServicesChosenSpace, in
      // that case, resultSpace will return with the users' chosen space:
      ioCSinfo->sourceSpace = plugIncolorServicesRGBSpace;
      ioCSinfo->resultSpace = plugIncolorServicesChosenSpace;
```

```
        // Private data. Must be NULL or you'll receive an error:
        ioCSinfo->reservedSourceSpaceInfo = NULL; // Must be NULL.
        ioCSinfo->reservedResultSpaceInfo = NULL; // Must be NULL.
        ioCSinfo->reserved = NULL; // Must be NULL.

        // Typical operation is choose a color or show the picker:
        ioCSinfo->selectorParameter.pickerPrompt = NULL;

        // Size parameter easily filled by taking size of structure:
        ioCSinfo->infoSize = sizeof(*ioCSinfo);
    }
    else
    {
        // You have to pass a ColorServicesInfo struct pointer. If you didn't,
        // this function returns a missing parameter error:
        err = errMissingParameter;
    }

    return err;

} // end CSInitInfo
```

`CSInitInfo` provides a function to reset and validate your initial `ColorServicesInfo` struct. Then you can set it up to tell the Color Services routine to grab a sample point. Figure 3 is a routine to return the first point in a window, in RGB space (regardless of what space the image is in).

Figure 3: GetSamplePoint routine

```
void GetSamplePoint (  globals,
                       *outSampleColorArray)
{
    OSErr err = noErr; // Assume no error, initially.

    // First make sure the ColorServices suite is available, and pop an alert if not:
    if (WarnColorServicesAvailable())
    { // was available. Go ahead and do this.

        // (1) Create the structure variable and the sample point location:
        ColorServicesInfo csInfo;

        // (2) Initialize the structure and check for any errors:
        err = CSInitInfo(&csInfo);

        if (err == noErr)
        {
            // No errors. Do this.
```

```
         // Lets go ahead and create a sample point to use. This can
         // be whatever you need, including examining the image’s
         // size and using values from that:
         Point myPoint = { 1, 1 };

         // (3) Override the default selector with the sample point command:
         csInfo.selector = plugIncolorServicesSamplePoint;

         // (4) Our source space has been inited to RGBColor.
         // So now, just set our selectorParameter to sample:
         csInfo.selectorParameter.globalSamplePoint = &myPoint;

         // (5) Call the proc using the macro from PIUtilities:
         // #define ColorServices(info) (*(gStuff->colorServices)) (info)
         err = ColorServices(&csInfo);

         if (err == noErr)
         {
            // Got the sample point. Return its data.

            // *outSampleColorArray is simply an array of four int16s.
            // Lets use the PIUtilities CSCopyColor routine to copy the
            // returned values into the outgoing array:
            CSCopyColor(colorComponents, outSampleColorArray);
            // outSampleColorArray[0] = red,
            // outSampleColorArray[1] = green,
            // outSampleColorArray[2] = blue,
            // outSampleColorArray[3] = undefined.
         }

      } // err

   } // ColorServices available

   return err;

} // end GetSamplePoint
```

`CSCopyColor`, called from `GetSamplePoint` is defined in `PIUtilities` and broken out in Figure 4.

Figure 4: CSCopyColor routine

```
OSErr CSCopyColor (int16 *outColor, const int16 *inColor)
{
   short loop;
   OSErr err = noErr;
   if (outColor != NULL && inColor != NULL)
   {
      for (loop = 0; loop < 4; loop++)
      {
         outColor[loop] = inColor[loop];
      }
   }
   else
   {
      // If the function was passed bad pointers, it will return
      // a missingParameter error:
      err = errMissingParameter;
   }

   return err;

} // end CSCopyColor
```

`CSCopyColor`, in other words, is a fairly straight-forward copy routine from one four-element `int16` array to another.

Returning the foreground or background color

In many cases, returning the foreground or background color from the Color Services suite is harder than just grabbing it from the parameter block. For instance, the Filter parameter block has two parameters defined for the background and foreground colors, in the color space native to the image:

```
   FilterColor          backColor;
   FilterColor          foreColor;
```

See `PIFilter.h` for more details.

Some parameter blocks, however, such as the Export parameter block, do not provide values for the foreground and background colors. In that case you'd want to use the Color Services suite to provide that information. Here's a sample routine to return the Foreground and Background colors in two color arrays:

Figure 5: GetForeAndBackColors routine

```
void GetForeAndBackColors (   globals,
                              *outForeColorArray,
                              *outBackColorArray)
{
   OSErr err = noErr; // Assume no error, initially.
```

```
// First make sure the ColorServices suite is available, and pop an alert if not:
if (WarnColorServicesAvailable())
{  // Available. Go ahead and do this.

   // (1) Create the structure variable and the sample point location:
   ColorServicesInfo csInfo;

   // (2) Initialize the structure and check for any errors:
   err = CSInitInfo(&csInfo);

   if (err == noErr)
   {  // No errors. Do this.

      // (3) Override the default selector with the get special color command:
      csInfo.selector = plugIncolorServicesGetSpecialColor;

      // (4) Our source space is inited to RGBColor.
      // So now, just set our selectorParameter to one of the colors:
      csInfo.selectorParameter.specialColorID =
         plugIncolorServicesForegroundColor;

      // (5) Call the proc using the macro from PIUtilities:
      // #define ColorServices(info) (*(gStuff->colorServices)) (info)
      err = ColorServices(&csInfo);

      if (err == noErr)
      {
         // Got the sample point. Return its data.

         // *outForeColorArray is simply an array of four int16s.
         // Lets use the PIUtilities CSCopyColor routine to copy the
         // returned values into the outgoing array:
         CSCopyColor(colorComponents, outForeColorArray);

         // (6) Now lets get the background color:
         csInfo.selectorParameter.specialColorID =
            plugIncolorServicesBackgroundColor;

            err = ColorServices(&csInfo);

            if (err == noErr)
               CSCopyColor(colorComponents, outBackColorArray);

      } // err @ ColorServices

   } // err @ CSInitInfo

} // ColorServices available

return err;
```

```
} // end GetForeAndBackColors
```

The colors will be returned with the array, in the case of RGB, initialized to:

1. array[0] = red,
2. array[1] = green,
3. array[2] = blue, and
4. array[3] = undefined.

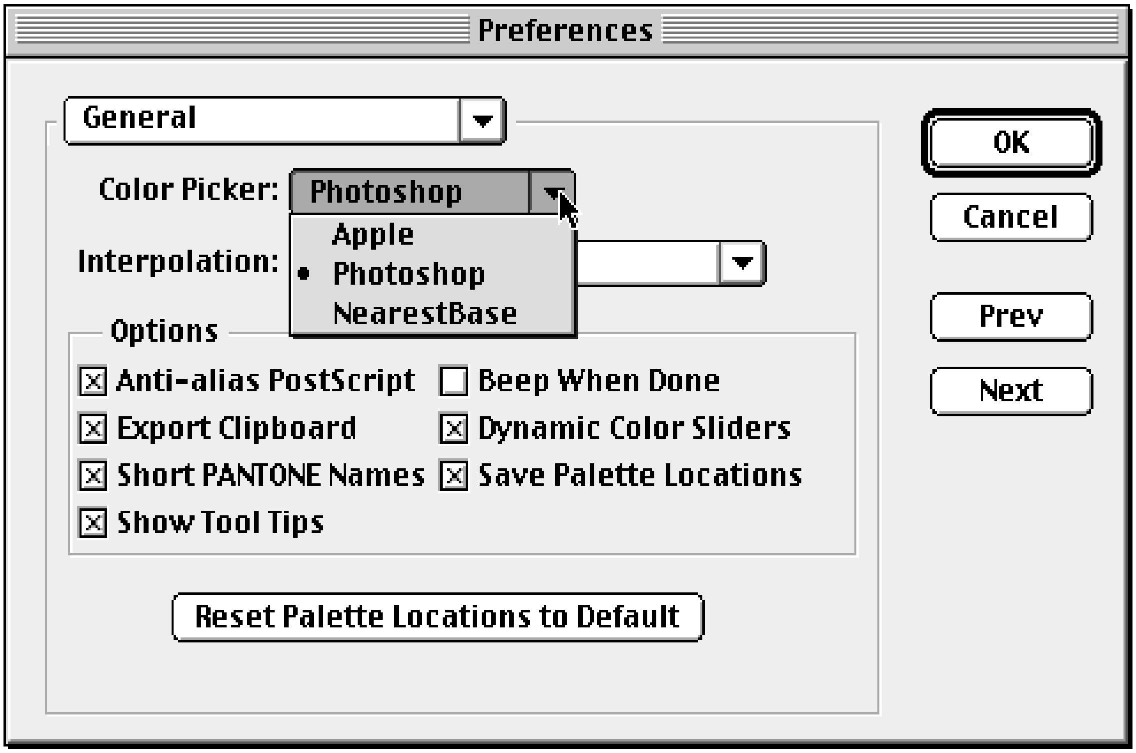
Each element is a number from 0 to 255, 255 being the most saturated and 0 indicating an absence of that color.

The parameters of the different color spaces will be discussed later, when converting from one color space to another. First, let’s look at how a user would pick a new color.

Choosing a color

There are times when you'll want a user to pick a color so that your filter or similar plug-in can use it for an effect. The Color Services suite provides a selector, `plugIncolorServicesChooseColor`, which pops the users' chosen color picker. The user chooses a color picker via the **File** » **Preferences** » **General** preference panel.

Figure 6: Selecting a color picker from the Photoshop General Preferences Panel



General

Color Picker: Photoshop

Interpolation:

☒ Anti-alias PostScript
☐ Beep When Done
☒ Export Clipboard
☒ Dynamic Color Sliders
☒ Short PANTONE Names
☒ Save Palette Locations
☒ Show Tool Tips

In this panel the user can choose between the OS color picker (Windows or Apple), the Photoshop color picker, and any plug-in Color Picker modules. In the example in Figure 1, there is a plug-in color picker called "NearestBase" which is the additional selection.

The examples in this article show the default, the Photoshop color picker.

Now all you need is a routine to pop that picker and return the color. There's already one in `PIUtilities` with its cover macro, `CSPickColor`. `CSPickColor` pops the color picker and returns the users' chosen space and color. You could force `CSPickColor` to always return a specific space, but it's preferable to be flexible and convert the color to the target space after its returned.

`CSPickColor` takes a color space and color as input, since the picker does have to default to something in its initial dialog settings. Another input value is the prompt string which is displayed at the top of the picker.

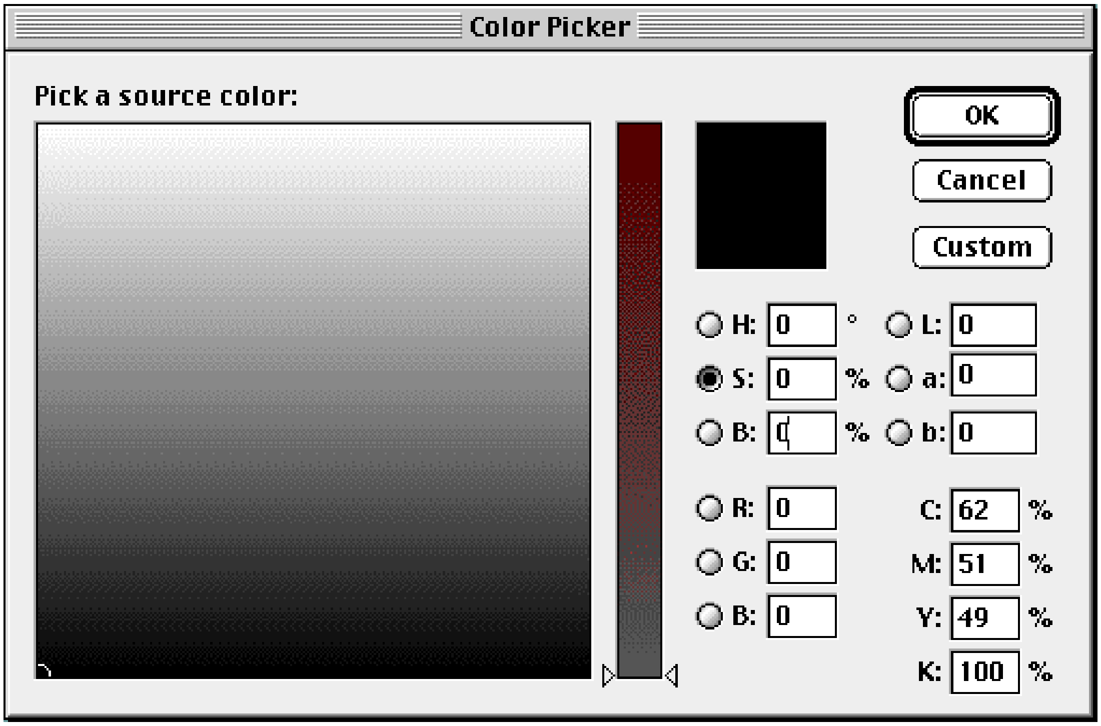
Figure 7: CSPickColor routine

```
OSErr HostCSPickColor ( ColorServicesProc proc,
                        const Str255 inPrompt,
                        int16 *ioSpace,
                        int16 *ioColor)
{
   OSErr err = noErr; // Assume no error, initially.

   if (HostColorServicesAvailable(proc))
   {  // Color Services are available. Now populate color services info with stuff.

      if (ioColor != NULL && ioSpace != NULL)
      {  // Parameters good.

         ColorServicesInfo                csinfo;

         // Initialize our info structure with default values:
         err = CSInitInfo(&csinfo);

         if (err == noErr)
         { // Init went fine. Now override default values with our own:

            // (1) Set selector to choose color and prompt string:
            csinfo.selector = plugIncolorServicesChooseColor;
            csinfo.selectorParameter.pickerPrompt = (Str255 *)inPrompt;

            // (2) Set initial color components in source:
            csinfo.sourceSpace = *ioSpace;
            CSCopyColor (csinfo.colorComponents, ioColor);

            // (3) Call convert routine with this info:
            err = (*proc)(&csinfo);

            // (4) If no error, copy the converted colors and user-picked
            // result space:
            if (err == noErr)
            {
               CSCopyColor (ioColor, csinfo.colorComponents);
               *ioSpace = csinfo.resultSpace;
            } // copy

         } // initinfo
```

```
        } // outColor and outSpace

        else
        { // outColor or outSpace pointer bad.
            err = errMissingParameter;
        }
    }
    else
    { // color services suite was not available
        err = errPlugInHostInsufficient;
    }

    return err;

} // end HostCSPickColor
```

Calling `CSPickColor` will show a color picker such as in Figure 8.

Figure 8: Photoshop Color Picker



○ H: 0 °
● S: 0 %
○ B: 0 %
○ L: 0
○ a: 0
○ b: 0
○ R: 0
○ G: 0
○ B: 0
C: 62 %
M: 51 %
Y: 49 %
K: 100 %

In the case of Figure 8, the users’ choices in the Color Picker reflect, basically, that the user picked nothing or reset everything to 0. Therefore, the return color will be {`0`, `0`, `0`, *undefined*} with a space of `plugIncolorServicesHSBSpace`. Let's assume, in this case, that you'll want to convert that color to the current color space of the document.

Converting between color spaces

What if you want to ask the user for a color then convert it into the document's color space? In a Filter plug-in, you can get the document's color mode from the parameter `imageMode`, which will be one of the supported document image modes.

Once you know the image mode, you will want to find that same spaces' entry for the Color Services suite. There is a facility in `PIUtilities` to do just that: `CSModeToSpace`.

Figure 9: CSModeToSpace routine

```
int16 CSModeToSpace (int16 imageMode)
{
   // Default to same space:
   int16 space = plugIncolorServicesChosenSpace;

   if (imageMode >= plugInModeBitmap && imageMode <= plugInModeRGB48)
   {
      /* static */ const int16 modePerSpace [] =
      {
         // Little array to map modes to color space values.  Make this static if you have
         // A4-global space set up (Macintosh 68k) and this will run faster.
         // If you need A4-space and don't have it set up, and this is made
         // static, you'll have garbage next time it's used:

         /* plugInModeBitmap */          plugIncolorServicesChosenSpace,
         /* plugInModeGrayScale */       plugIncolorServicesGraySpace,
         /* plugInModeIndexedColor */    plugIncolorServicesChosenSpace,
         /* plugInModeRGBColor */        plugIncolorServicesRGBSpace,
         /* plugInModeCMYKColor */       plugIncolorServicesCMYKSpace,
         /* plugInModeHSLColor */        plugIncolorServicesHSLSpace,
         /* plugInModeHSBColor */        plugIncolorServicesHSBSpace,
         /* plugInModeMultichannel */    plugIncolorServicesChosenSpace,
         /* plugInModeDuotone */         plugIncolorServicesGraySpace,
         /* plugInModeLabColor */        plugIncolorServicesLabSpace,
         /* plugInModeGray16 */          plugIncolorServicesGraySpace,
         /* plugInModeRGB48 */           plugIncolorServicesRGBSpace
      };

      space = modePerSpace[imageMode];

   } // If unsupported mode, will return current space.
   return space;

} // end CSModeToSpace
```

The call to `CSModeToSpace` is straightforward:

```
   int16 currentSpace = CSModeToSpace(gStuff->imageMode);
```

The next step is converting from a given space to another. `PIUtilities` contains another routine, `HostCSConvertColor`, with its accompanying macro that you'll want to use to call the routine, `CSConvertColor`, takes a source space, a result space, and a color array, and returns the result space in the color array, or an error if unable to convert the color or missing a parameter.

Figure 10: CSConvertColor routine

```
OSErr HostCSConvertColor (    ColorServicesProc proc,
                              const int16 sourceSpace,
                              const int16 resultSpace,
                              int16 *ioColor)
{
   OSErr err = noErr;

   if (HostColorServicesAvailable(proc))
   { // Color Services available. Now populate color services info with stuff:
      if (ioColor != NULL)
      { // Parameter good.

         ColorServicesInfo                csinfo;

         // Initialize our info structure with default values:
         err = CSInitInfo(&csinfo);

         if (err == noErr)
         { // Init'ed okay.

            // Now override default values with our own.

            // (1) Copy original to colorComponents space then
            // set up space to convert from source to result:
            err = CSCopyColor(&csinfo.colorComponents[0], ioColor);

            if (err == noErr)
            { // Copied okay.
               // (2) Set the source and result space:
               csinfo.sourceSpace = sourceSpace;
               csinfo.resultSpace = resultSpace;
               // (3) Call the convert routine with this info:
               err = (*proc)(&csinfo);

               // (4) If no error, copy the converted colors:
               if (err == noErr)
                  CSCopyColor (ioColor, csinfo.colorComponents);

            } // copy

         } // initinfo

      } // ioColor

      else
      { // ioColor pointer bad.
         err = errMissingParameter;
      }
   }
   else
   { // Color services suite was not available.
      err = errPlugInHostInsufficient;
   }

   return err;

} // end HostCSConvertColor.
```

So to pick then convert the color to the document’s color space:

Figure 11: PickThenConvertColor routine

```
void PickThenConvertColor(GPtr globals)
{
   OSErr err = noErr;

   // Define some initial values for the picker to display:
   int16          ioColors[4];
   int16          ioSpace = plugIncolorServicesHSBSpace;

   // Prompt string must be a Pascal string. So, for Windows, do:
   #ifdef __PIWin__
      Str255      prompt = "\x0CPick a color";
   #else // Mac and Unix understand \p:
      Str255      prompt = "\pPick a color";
   #endif

   // Set default HSB color. 1=Hue, 2=Saturation, 3=Brightness, 4=undefined:
   CSSetColor(ioColors, 0, 255, 255, 0);
   err = CSPickColor(prompt, &ioSpace, ioColors);
```

```
    if (err == noErr)
    { // Picked the color successfully. Now convert it to the right space:
        int16       resultSpace = CSModeToSpace(gStuff->imageMode);

        CSConvertColor (ioSpace, resultSpace, ioColors);

        // Now ioColors contain the colors in the documents color mode.
        // Do something cool with them here.
    }
    else
    {
        // Some error occurred while picking. Examine and do something
        // with it here.
    }
} // end PickThenConvertColor.
```

Figure 12 is the values of the color components of the different color spaces. It is excerpted from the Photoshop SDK Guide from Table B-1. Refer to the Guide for more information.

Figure 12: colorComponents array structure

Color space	color Components[0]	color Components[1]	color Components[2]	color Components[3]
RGB	red from 0...255	green from 0...255	blue from 0...255	undefined
HSB	hue from 0...359 degrees	saturation from 0...255 representing 0%...100%	brightness from 0...255 representing 0%...100%	undefined
CMYK	cyan from 0...255 representing 100%...0%	magenta from 0...255 representing 100%...0%	yellow from 0...255 representing 100%...0%	black from 0...255 representing 100%...0%
HSL	hue from 0...359 degrees	saturation from 0...255 representing 0%...100%	luminance from 0...255 representing 0%...100%	undefined

Figure 12: colorComponents array structure (Continued)

Color space	color Components[0]	color Components[1]	color Components[2]	color Components[3]
Lab	luminance value from 0...255 representing 0...100	a chromanance from 0...255 representing –128...127	b chromanance from 0...255 representing –128...127	undefined
Gray scale	gray value from 0...255	undefined	undefined	undefined
XYZ	X value from 0...255	Y value from 0...255	Z value from 0...255	undefined

Gotchas

There are some gotchas in the Color Services suite that have resulted in less use over the last years. One is the inability to convert arrays of spaces.

Another difficulty is a legacy bug where no matter what space you convert to, the proc always returns `paramErr` (-50) and converts the requested color to RGB. This has been fixed since Photoshop 3.0.4.

The other issue with the Color Services routine is that converting from some spaces to others has not received consistent feedback from third-party developers regarding its validity. Particularly, Lab and XYZ space conversions have been questioned. Unfortunately, those spaces are defined differently in multiple reference guides, and, where one developer might report bad output, another reports those same numbers are exactly what is expected from that space. (Lab, for instance, is based on visual acuity readings of a human subject in a specified environment. A reading that is made to be for the “average” human, with plenty of delta.) Due to that, and time constraints, engineering has only been able to verify the top used spaces. You can, with confidence, use and convert between any of these spaces:

> RGB, HSB, CMYK, Gray, HSL

The other’s *may* be suspect. Try them and see if you get results consistent with your expectations. Let the ADA know if you don’t. Especially, let the ADA know what you *expected* if it differs from what you got.

Color Picker plug-ins

With the Color Services suite now under your belt, Color Picker plug-ins should be much more understandable. A Color Picker plug-in's job is to create its own User Interface (UI) to allow the user to pick a color, then return that color in the same color array comprised of four unsigned 16-byte integers, and an enum for what space the result is in.

Color space required for returning a color

While you can implement your own UI to allow a user to pick a color in any space (such as six- and eight-value array spaces, as are becoming popular), you must return a color in the known array space.

This makes sense since the definitons for the different plug-in color services spaces are Photoshop defined, not plug-in defined. There would be no way for a caller to know you're returning a color in six-value space, for instance. So you need to convert that color back into 4-value space, and assign it a space in the Color Services color space enumerations.

Structure for passing color in and out of a picker

The Color Picker plug-in has a simple parameter block that includes a `PickParms` structure that is defined as:

Figure 13: PickParms structure

```
typedef struct PickParms
{
   int16 sourceSpace;      // The colorspace the original color is in
   int16 resultSpace;      // The colorspace of the returned result
                           // Can be plugIncolorServicesChosenSpace.
   // The original color on entry, the returned color on exit:
   unsigned16 colorComponents[4];

   // Pointer to prompt string:
   Str255 *pickerPrompt;

} PickParms;
```

When your Color Picker plug-in is first called, this structure will contain the incoming information for its source space, requested result space, source color, and picker prompt string. You can choose to use or ignore any of those values (for instance, if your plug-in does not need a prompt string).

Color Picker plug-in selectors

The Color Picker plug-in module only gets two selectors, vastly simplifying the dispatching mechanism. They are:

```
pickerSelectorAbout (0)
pickerSelectorPick (1)
```

Obviously, you want to pop your about box on `pickerSelectorAbout`, and pop your UI to pick a color on `pickerSelectorPick`.

Returning a color

It's your responsibility to populate the `PickParms` structure with any new color or space that the user has selected. If you don't do anything, the source color will be returned in the result space.

At minimum, you must copy `resultSpace` to `sourceSpace`. This is because your plug-in may be called with `resultSpace` equalling `plugIncolorServicesChosenSpace`, the command that tells your plug-in to return `resultSpace` as the users' selected space.

`plugIncolorServicesChosenSpace` makes no sense as a `resultSpace` once execution is returned from your plug-in, especially if it returns from the picker command with no error. So, at least, do:

```
gStuff->pickParms.sourceSpace =
   gStuff->pickParms.resultSpace

return;
```

Popping your UI

Your user interface will be a platform-dependent modal dialog box. Any thermometer or other active tools you'll have to implement yourself. Make sure you don't call the Color Services with a `plugIncolorServicesChooseColor` selector, as that will cause an endless loop!

Example color picker main routine

Here's a sample picker routine from the plug-in example *NearestBase* from the Photoshop SDK. NearestBase simply cycles through four solid base colors and returns the result in CMYK space:

Figure 14: DoPick routine

```
void DoPick (GPtr globals)
{

   // Set result space to CMYK (see PIGeneral.h for valid spaces):
   gStuff->pickParms.resultSpace = plugIncolorServicesCMYKSpace;

   switch (gCount % 4)
   { // Cycle through four cases:

      case 0: // red
         gStuff->pickParms.colorComponents[0] = 255 * 0x101;
         gStuff->pickParms.colorComponents[1] = 0 * 0x101;
         gStuff->pickParms.colorComponents[2] = 0 * 0x101;
         gStuff->pickParms.colorComponents[3] = 255 * 0x101;
         break; // 0

      case 1: // green
         gStuff->pickParms.colorComponents[0] = 0 * 0x101;
         gStuff->pickParms.colorComponents[1] = 255 * 0x101;
         gStuff->pickParms.colorComponents[2] = 0 * 0x101;
         gStuff->pickParms.colorComponents[3] = 255 * 0x101;
         break; // 1

      case 2: // blue
         gStuff->pickParms.colorComponents[0] = 0 * 0x101;
         gStuff->pickParms.colorComponents[1] = 0 * 0x101;
         gStuff->pickParms.colorComponents[2] = 255 * 0x101;
         gStuff->pickParms.colorComponents[3] = 255 * 0x101;
         break; // 2

      case 3: // black
         gStuff->pickParms.colorComponents[0] = 0 * 0x101;
         gStuff->pickParms.colorComponents[1] = 0 * 0x101;
         gStuff->pickParms.colorComponents[2] = 0 * 0x101;
         gStuff->pickParms.colorComponents[3] = 0 * 0x101;
         break; // 3

      default: // Unsupported, "exercise" error message.
         gResult = pickerBadParameters;
         break;

   } // Switch gCount.

   gCount++;

} // end DoPick.
```

Color Picker Caveats

There are some things you'll want to watch out for when you write your own Color Picker.

Color Pickers are plug-ins called by plug-ins

Whenever the user clicks on a swatch or a foreground or background color, the chosen Color Picker is displayed. This means your plug-in's UI is called, if the user has selected it in the General Preferences window.

Your Color Picker plug-in, however, can also be called from another plug-in. As you've seen in the previous section on the Color Services suite, a plug-in developer can ask the suite to pop the current Color Picker and return a color. This can get you into a precarious position, having one plug-in being called from another. Here are some things to watch out for, in that event:

1. On the Macintosh, if you are working with resources you will have a couple extra resource forks open besides the application's. You'll have the calling plug-in's resource fork open, as well as your Color Picker plug-in's. Make sure if you're using string, icon, or any other resources that you use the toolbox call `Get1Resource()` instead of `GetResource`, which will use the currently active resource and drop through the resource chain, instead of just searching one resource deep.
2. Photoshop is memory intensive, and it's more than likely that your Color Picker plug-in is being called from a memory-intensive plug-in, such as a Filter plug-in. Be as conservative with your memory usage as possible, and mind the `maxData` value carefully and stick to the rule of only using half of it and leaving half of it free for transient operations.
3. The Color Services suite is still available to you to do any sort of conversions, sample grabbing, or special color returning. This makes sense if you want to paint on your UI then grab sample colors from it. However, do not call the Color Services suite with `plugIncolorServicesChooseColor`. Since you're already the active Color Picker, you may have been invoked by the Color Services suite in the first place. Calling it again to choose a color will result in an endless loop.

Other issues and future implementation

Support for 16bpc color

16bpc, stands for 16-bits-per-channel color, which is supported in some of the modes such as `RGB48` and `GrayScale16`.

As Photoshop matures its color services as a whole, you'll see more deep color references and additional functionality to deal with those new color spaces. Adobe is very committed to pushing all their applications to include more deep color, and similar critical color features, with every new version.

Next issue

Next issue we'll delve into the new selection modules and the ChannelPorts callback suite, and what it took to make a plug-in that could return a path in the shape of a musical note.

###