Color Programming Topics for Cocoa

Cocoa > Graphics & Imaging

Contents

Figures, Tables, and Listings

Introduction to Color Programming Topics for Cocoa

The color classes let you specify colors and let users choose colors.

Organization of This Document

This topic contains the following articles:

- "About Color Spaces" (page 9) describes what color spaces are and which ones are available in the Application Kit.

- "About Color Lists" (page 17) describes what color lists are and how they're used.

- "Working With Color Spaces" (page 19) discusses how you can create and convert colors using color-space objects and names and how you can create custom color spaces.

- "Accessing a Color's Components" (page 25) describes how to retrieve a color's individual components.

- "Accessing System Colors" (page 27) describes how to access colors that are controlled by user preferences.

- "Using the System Control Tint" (page 29) describes how to use the system-wide control tint in your custom views and control cells.

- "Choosing Colors With Color Wells and Color Panels" (page 33) describes how to let users select colors using color wells and the color panel.

- "Choosing the Color Pickers in a Color Panel" (page 35) describes how to choose which color pickers are available in the color panel and which one is selected.

- "Adding Custom Color Pickers to a Color Panel" (page 37) describes how to add color pickers to the color panel.

- "Subclassing NSColor" (page 45) describes how to create custom subclasses of NSColor.

- "Storing NSColor in User Defaults" (page 47) describes how to store colors in an application's user defaults.

About Color Spaces

A color space describes an abstract, multidimensional environment in which any particular color can be defined. The following sections summarize the basic concepts and terminology of color spaces and discusses how Cocoa implements them.

Some of the information presented here is adapted from *Color Management Overview*. For a thorough description of color and color spaces, see that document.

Color Models and Color Spaces

The human eye apprehends color as light in a fairly narrow band of the electromagnetic spectrum. The biology of the eye makes it particularly receptive to red, blue, and green light. Humans can visualize a broad range of colors through mixtures of these three primary colors.

A color model is a geometric or mathematical framework that attempts to describe the colors we see. It uses numerical values pinned to dimensions of the model to represent the visible spectrum of color. A color model gives us a method for describing, classifying, comparing, and ordering colors.

A color space is a practical adaptation of a color model that specifies a gamut of colors that can be produced using that model. The color model determines the relationship between values, and the color space defines the absolute meaning of those values as colors. These values, called components, are in most instances floating-point values between 0.0 and 1.0.

Gray, RGB, and CYMK Color Spaces

The simplest color space is the gray space (sometimes also called the white space). The gray space has a single dimension or component, ranging from pure white to pure black; it is used for grayscale printing.

RGB is a three-dimensional color model whose name (as with most color spaces and color models) represents its components—in this case red, green, and blue. RGB-based color spaces are additive, meaning that the three primary colors red, green, and blue are added together in various proportions of intensity to create the colors of the visible spectrum. RGB color spaces are used for devices such as color displays and scanners.

On the other hand, color spaces based on the CYM color model are subtractive. The letters in the model name stand for the components cyan, yellow, and magenta. The major color space based on CYM is CYMK; the "K" in its name stands for the key color, which is black. The subtractive color theory, which underlies CYM, holds that various levels of cyan, magenta, and yellow absorb or "subtract" a portion of the spectrum of the white light illuminating an object. The color of an object is the result of the lights that are not absorbed by the object. The black in the CYMK color space is used to compensate for the interaction of the three primary colors on white paper. The CYMK color space is most commonly used for color printers and similar output devices.

As Figure 1 illustrates, the RGB and CYM color models are complementary, with one being additive and the other subtractive (the cyan corner in this model representation is hidden from view).

Figure 1   The RGB and CYM color models
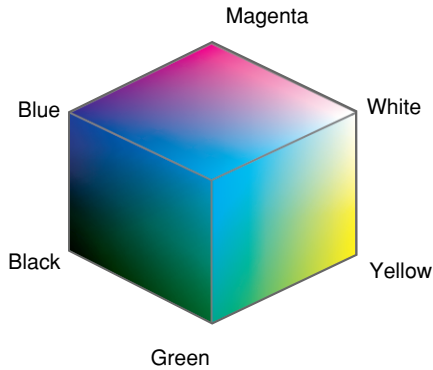
Two important and related transformations of the RGB color model are the HSV and HLS color spaces. Instead of making red, green, and blue the operative components of the space, these spaces describe colors in terms more natural to an artist:

■   HSV—hue, saturation, value (also known as HSB, where "B" represents brightness)

■   HLS—hue, lightness, saturation

The HSV/B and HLS spaces use models that assign values to these components in conical geometries, as illustrated in Figure 2.

Figure 2   The HSV and HLB color models
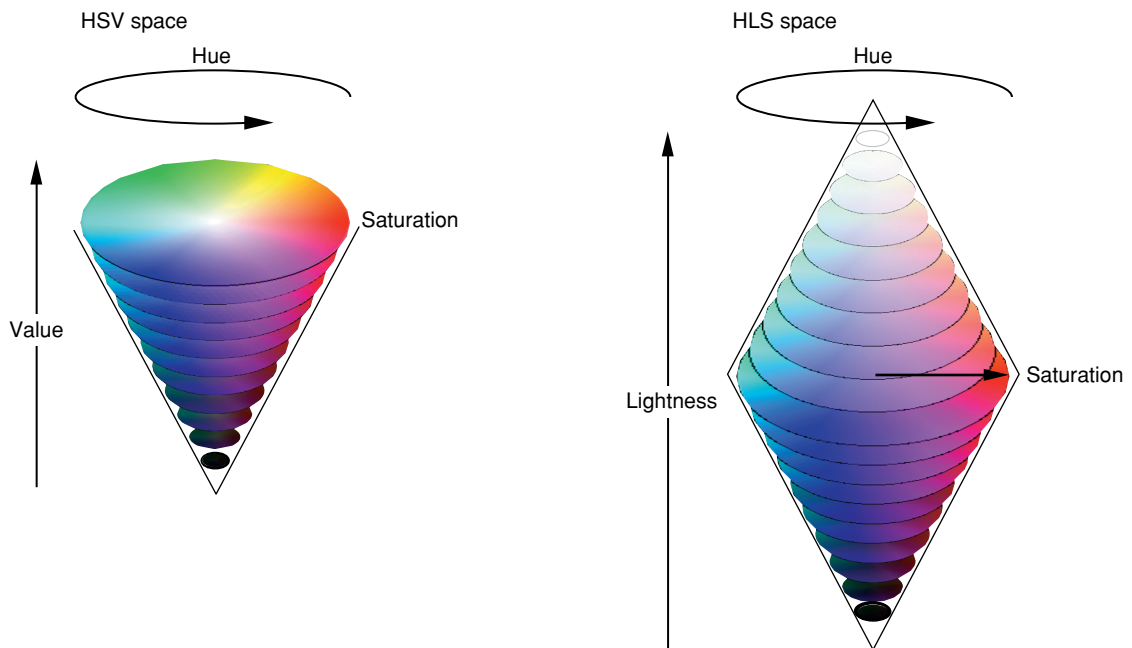
The hue component in both spaces is a measurement in degrees of color in a spectrum formed into a circle. The values are incremented in a counterclockwise direction: a hue value of zero specifies red, a hue value of 120 indicates green, and so on. In both the HSB and HLS spaces, the saturation component measures color intensity (making the major difference, for example, between tan and brown). The lightness and value (or brightness) components of the different spaces are almost identical. They measure the absence of light—or black—that is part of particular color.

The color panel used in Mac OS X applications has a color-wheel pane that simulates the HSB model (Figure 3).

Figure 3   The color-wheel pane of the color panel
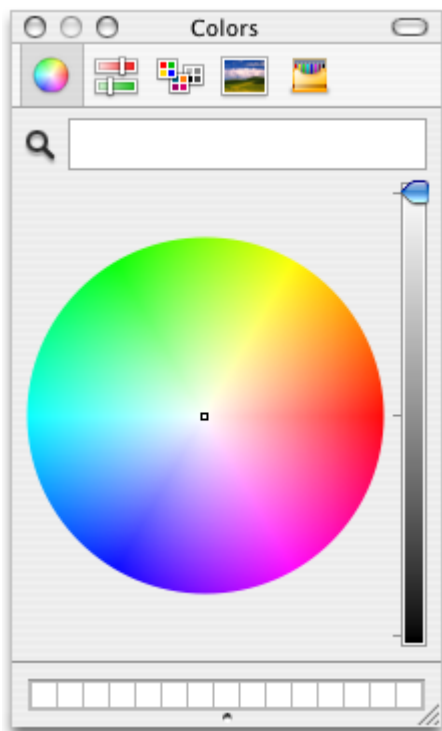
Device-Independent Color Spaces

Color spaces based on RGB and CYM color models can be device-dependent or device-independent. Colors from device-dependent color spaces are dependent on the physical characteristics of devices such as monitors (RGB and grayscale) and printers (CYMK) as well as the properties of materials such as ink and paper. Even the age of a device can affect the color it produces. Device-dependent color spaces are limited by the gamut, or range, of colors that a particular device is capable of. Consequently, colors in a device-dependent color space can appear different when rendered by different devices of the same general type.

One can also note subtle color differences among color spaces in the same color-model "family." For example, the RGB color model has many RGB color spaces, such as ColorMatch, Adobe RGB, sRGB, and ProPhoto RGB. You can assign the same RGB component values to profiles that describe these different RGB color spaces. The color from each color space looks different when rendered, but the numeric values and model are the same.

Some color spaces can express color in a way that is independent of any device. The colors of these device-independent color space are more accurate representations of the colors perceived by the human eye. They derive from the response of the retina to the three primary stimuli of visible light. Many device-independent color spaces result from work carried out by the Commission Internationale d'Eclairage (CIE) and for that reason are also called CIE-based color spaces. Three of the more important CIE-based spaces are XYZ, Yxy, and L*a*b*. Figure 4 depicts the L*a*b* color space.

Figure 4   The L*a*b* color space

L*a*b* color space
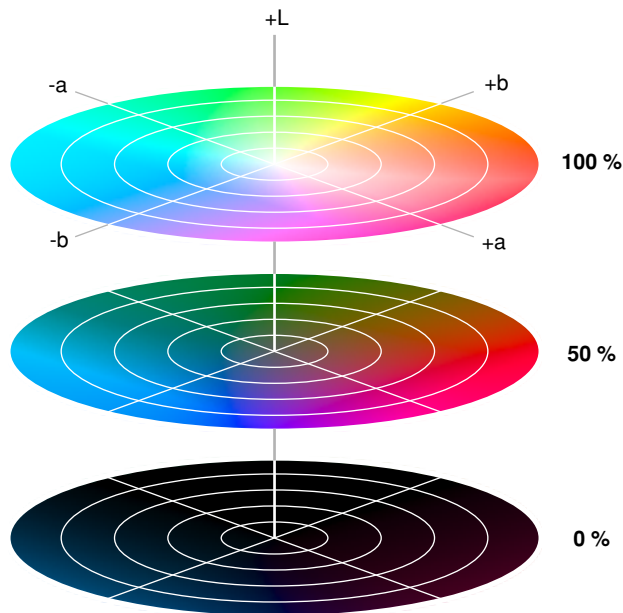
One important use for device-independent color spaces is to convert a color in one device-dependent color space to a reasonably approximate color in a different device-dependent color space. For example, if a program wanted to ensure that a photo displayed on a color monitor (using a RGB color space) was accurately rendered on a printer (using a CYMK color space), it might use a device-independent color space as an interchange space.

Color Spaces in Cocoa

The Application Kit represents color spaces in two ways: through color-space names and color-space objects.

Color-Space Names

Color-space names are global string constants declared in `NSGraphics.h` that designate predefined color spaces. You can use a color-space name in certain methods of NSColor that create or convert color objects. The name identifies the color space to be used for the operation. Table 1 lists the currently defined constants.

Table 1   Color-space names in the Application Kit

Color-space Name	Description
`NSDeviceCYMKColorSpace`	Device-dependent color space with cyan, magenta, yellow, black, and alpha components
`NSDeviceWhiteColorSpace`	Device-dependent color space with white and alpha components (pure white is 1.0)
`NSDeviceBlackColorSpace`	Device-dependent color space with black and alpha components (pure black is 1.0)
`NSDeviceRGBColorSpace`	Device-dependent color space with red, green, blue, and alpha components. However, you can also create a color with HSB (hue, saturation, brightness) and alpha components and can extract these components.
`NSCalibratedWhiteColorSpace`	Calibrated color space with white and alpha components (pure white is 1.0)
`NSCalibratedBlackColorSpace`	Calibrated color space with black and alpha components (pure black is 1.0)
`NSCalibratedRGBColorSpace`	Calibrated color space with red, green, blue, and alpha components. However, you can also create a color with HSB (hue, saturation, brightness) and alpha components and can extract these components.
`NSNamedColorSpace`	Catalog name and color name components
`NSPatternColorSpace`	Pattern image (tiled)
`NSCustomColorSpace`	Custom NSColorSpace object and floating-point components describing a color in that space

The alpha component belongs to all color spaces used in Cocoa. It defines the opacity of a color; an alpha value of 1.0 indicates a completely opaque color and 0.0 indicates a completely transparent one.

The "Device" color-space names represent color spaces in which component values are applied to devices as specified. There is no optimization or adjustment for differences between devices in how they render colors. If you know exactly which device is connected to a system and you want to print or display a certain color on that device, then it makes sense to use a appropriate device-dependent color space when creating NSColor objects. However, it is usually not the case that an application knows which devices are connected and their specific color spaces. If you specify components of a color in a device-dependent color space—let's say `NSDeviceRGBColorSpace`—and then have several displays render this color, you will see several slightly different colors.

To get around this problem you can use calibrated color spaces, which are designated by two of the color-space names in Table 1. A calibrated color space is a device-independent color space. The color spaces designated by `NSCalibratedWhiteColorSpace` and `NSCalibratedRGBColorSpace` color spaces are *calibrated* to a device that best represents devices in a particular class, such as color displays. It allows your application to present reasonably accurate colors when you are unsure about the color space of a device in a particular context.

The color-space name `NSNamedColorSpace` identifies a special type of color spaces. The components of this color space are indexes into lists or catalogs of prepared colors. The catalogs of named colors come with lookup tables that are able to generate the correct color on a given device.

> **Note:** Not all named color lists (that is, NSColorList objects) are catalogs; only the ones defined in the `NSNamedColorSpace` color space have associated lookup tables. In general, you do not create color catalogs at runtime with NSColor and NSColorList methods.

The `NSPatternColorSpace` color-space name identifies a pattern color space, which is simply an image that is repeated over and over again in a tiled pattern.

The `NSCustomColorSpace` color-space name identifies a custom NSColorSpace object. A custom color-space object represents a color space that is not necessarily predefined by the Application Kit. See "Making Custom Color Spaces" (page 21) for information on creating custom color-space objects.

Color-Space Objects

In Cocoa, objects can represent color spaces just as color-space names can designate them. In fact, each color-space name identifies an underlying color-space object created by the Application Kit. Color-space objects are instances of a class that inherits from the NSColorSpace class.

> **Important:** The NSColorSpace class was introduced in Mac OS X v10.4. Public color-space objects and the `NSCustomColorSpace` name discussed in the previous section are not available in earlier version of the operating system.

Most of the color-space names designate a color space represented by an underlying NSColorSpace object. These same objects are returned by class factory methods of NSColorSpace. Table 2 shows the correlation between color-space name and factory color-space object.

Table 2   Corresponding color-space names and factory methods of NSColorSpace

Color-space name	Color-space factory methods
`NSCalibratedWhiteColorSpaceNSCalibratedBlackColorSpace`	`genericGrayColorSpace`
`NSCalibratedRGBColorSpace`	`genericRGBColorSpace`
None	`genericCMYKColorSpace`
`NSDeviceWhiteColorSpaceNSDeviceBlackColorSpace`	`deviceGrayColorSpace`
`NSDeviceRGBColorSpace`	`deviceRGBColorSpace`
`NSDeviceCMYKColorSpace`	`deviceCMYKColorSpace`
`NSNamedColorSpace`	None
`NSPatternColorSpace`	None
`NSCustomColorSpace`	None

The NSColor class is the largest "client" of the NSColorSpace class. Many NSColor methods include a parameter for either specifying a color-space name or a color-space object. In fact, color objects cannot be created without an explicit or implicit reference to a color space, including named and pattern color spaces. (The color-creation methods of NSColor that don't specify a color space or color model in their names assume the calibrated RGB or calibrated white color spaces.) Other methods of the NSColor class allow you to convert a color object in one color space to an object representing a color in a different color space. For further information, see "Creating and Converting Colors Using Color Spaces" (page 19).

You can make custom color-space objects programmatically by using the NSColorSpace class. To create a custom color-space object you must initialize it with one of two sources of data:

- A ColorSync object—an object of opaque type `CMProfileRef`.

- ICC profile data—an NSData object encapsulating an ICC profile map; the map is a structure that consists of a header, a tag table, and tagged element data.

For more information on creating objects that represent custom color spaces, see "Making Custom Color Spaces" (page 21).

> **Note:** Although you can create custom NSColorSpace objects programmatically, there is usually little need to do so. The Color pane of the Display system preference lets you run a "wizard" for creating custom color spaces.

About Color Lists

An `NSColorList` is an ordered list of `NSColor` objects, identified by keys. Instances of `NSColorList`, or more simply, color lists, are used to manage named lists of color objects. The list-mode color picker of an `NSColorPanel` object uses instances of `NSColorList` to represent any lists of colors that come with the system, as well as any lists created by the user. An application can use `NSColorList` to manage document-specific color lists, which may be added to an application's `NSColorPanel` object using its `attachColorList:` method.

An `NSColorList` object is similar to a dictionary object: An color object is added to, looked up in, and removed from the list by specifying its key, which is a string object. These keys are used to identify the colors in the list and are used to display the color to the user in the color panel. In addition, colors can be inserted at specified positions in the list.

The color list has a name, specified when you create the object using either the `initWithName:` or `initWithName:fromFile:` method.

Instances of `NSColorList` are created for all user-created color lists (those in the color panel) and various color catalogs available on the system.

An `NSColorList` object saves and retrieves its colors from files with the extension ".`clr`" in directories defined by a standard search path. To access all the color lists in the standard search path, use the `availableColorLists` method; this returns an array of `NSColorList` objects, from which you can retrieve the individual color lists by name.

The standard search path for color lists is:

- `/System/Library/Colors`
- `/Local/Library/Colors`
- `~/Library/Colors`

The color lists returned by the `availableColorLists` method include color catalogs, made up of colors defined in the `NSNamedColorSpace` color space. One example is the System color list, which appears in the Mac OS X color panel under the name "Developer." Note, however, that not all named color lists are catalogs. In general, lists created at runtime with NSColor and NSColorList methods are not catalogs.

`NSColorList` reads color list files in several different formats; it saves color lists using the archiver API.

`NSColorList` posts an `NSColorListDidChangeNotification` when a color list is changed.

Working With Color Spaces

This document discusses the use of color spaces when creating or converting `NSColor` objects. It also describes how to create objects representing new color spaces by using the `NSColorSpace` class.

> **Important:** The `NSColorSpace` class and `NSColor` methods that take `NSColorSpace` objects as parameters are new with Mac OS X v10.4.

Creating and Converting Colors Using Color Spaces

The methods of the `NSColor` class that create or convert color objects make explicit or implicit reference to a color space, either in the form of a name or an object. After all, colors cannot be defined without a color space providing the environment for definition. The following sections summarize these `NSColor` methods and offer guidelines for their use in programming.

NSColor Methods That Use Color Spaces

Many color-creation factory methods of `NSColor` embed the name of the color space in the method name. You must specify the component values of the color after the keywords of the method. For example, `colorWithDeviceCyan:magenta:yellow:black:alpha:` creates the color using the color space designated by `NSDeviceCMYKColorSpace`. You would create a color in the calibrated HSB color space (which is designated by `NSCalibratedRGBColorSpace` in this case) using `colorWithCalibratedHue:saturation:brightness:alpha:`. Colors in named (catalog) and pattern color spaces also have their own factory creation methods: `colorWithCatalogName:colorName:` and `colorWithPatternImage:`

To create color objects in color spaces represented by `NSColorSpace` objects, use the `colorWithColorSpace:components:count:` method. As illustrated in Listing 1 (page 19), you can obtain one of the predefined color-space objects by invoking the appropriate `NSColorSpace` class factory method (for example, `genericCMYKColorSpace`).

Listing 1   Creating a color from a predefined color-space object

```
float comps[] = {0.4, 0.2, 0.6, 0.0, 1.0};
NSColor *aColor = [NSColor colorWithColorSpace:[NSColorSpace
genericCMYKColorSpace] components:comps count:5];
```

> **Note:** The `genericCMYKColorSpace` method of `NSColorSpace` is the only way to obtain a predefined calibrated CYMK color space; there is currently no corresponding color-space name.

The main advantage of using the `colorWithColorSpace:components:count:` method to create colors is that you can use an object representing a custom color space (see "Making Custom Color Spaces" (page 21)). You are not limited to the predefined color-space objects.

NSColorSpace objects have advantages over color-space names by virtue of being objects. For example, you can also query these objects about their properties, such as the number of components, the values of those component, and the localized name. You can also archive and unarchive color-space objects.

Some color-creation NSColor methods make no reference to a color space in their names. Some of these class factory methods create primary and secondary colors, such as `blueColor` and `purpleColor`; others, such as `lightGrayColor`, create grayscale colors. These `NSColor` factory methods assume a color space of calibrated RGB (`NSCalibratedRGBColorSpace`) or calibrated white (`NSCalibratedWhiteColorSpace`), as appropriate. In most cases the alpha component (opacity) is 1.0.

Other color-creation methods of `NSColor` create objects representing the standard colors of user-interface objects in Mac OS X; examples of these methods include `controlTextColor`, `gridColor`, and `windowFrameColor`. You should not make any assumptions about the color space of these colors. Indeed, for any color object, it is a good practice not to assume its color space, but instead to convert it to the desired color space before using it. See "Programming Guidelines for Color Spaces" (page 20) for more on this subject.

NSColor offers the `colorUsingColorSpaceName:` method for converting colors from once color space to another. This method takes a color-space name as an argument. Here's an example of a use of this method, which converts the `NSColor` object created above from a CMYK to an RGB color space:

```
NSColor *bColor = [aColor colorUsingColorSpaceName:NSCalibratedRGBColorSpace];
```

The color object created by this method usually has component values that are different from the original object's component values, but the colors look the same. Sometimes the color conversion is correct only for the current device; the method might even return `nil` if the requested conversion cannot be done.

Note that you should *not* use the `colorUsingColorSpace:` method for converting a color between color spaces. For example, the `NSColor` object created by the following method is not in the RGB color space and thus raises an exception when you send it a `redComponent` message:

```
NSColor *cColor = [aColor colorUsingColorSpace:[NSColorSpace
genericRGBColorSpace]];
```

Programming Guidelines for Color Spaces

Observe the following guidelines when dealing with NSColor objects and color spaces.

Which Color Space Do I Use?

The Cocoa color APIs give you a range of predefined color spaces to work with, either through color-space names or as objects returned from `NSColorSpace` factory methods. How do you know which color space to use in any given programming context?

Generally, it is recommended that you use calibrated (or generic) color spaces instead of device color spaces. The colors in device color spaces can vary widely from device to device, whereas calibrated color spaces usually result in a reasonably accurate color. Device color spaces, on the other hand, might yield better performance under certain circumstances, so if you know for certain the device that will render or capture the color, use a device color space instead.

As for the model of the predefined color space, it depends on where the color is to be rendered or captured. Use RGB for color monitors and scanners, `NSCalibratedWhiteColorSpace` or `genericGrayColorSpace` objects for grayscale monitors, and CYMK for printers. If the destination is indeterminate, use RGB.

Can I Access the Components of Any NSColor Object?

It is invalid to use an accessor method related to components of a particular color space on an `NSColor` object that is not in that color space. For example, `NSColor` methods such as `redComponent` and `getRed:green:blue:alpha:` work on color objects in the calibrated and device RGB color spaces. If you send such a message to an NSColor object in the CMYK color space, an exception is raised.

If you have an `NSColor` object in an unknown color space and you want to extract its components, you should first convert the color object to a known color space before using the component accessor methods of that color space. For example:

```
NSColor *aColor = [color colorUsingColorSpaceName:NSCalibratedRGBColorSpace];
if (aColor) {
    float rcomp = [aColor redComponent];
}
```

If the color is already in the requested color space, you get back the original object.

Making Custom Color Spaces

A developer rarely needs a color space that is not supplied by the system. Not only are the predefined generic (calibrated) and device color spaces sufficient for most purposes, but Mac OS X allows you to create custom color spaces without having to write any code. In the Color pane of the Displays system preference (Figure 1) there is a Calibrate button; clicking this button launches a wizard that steps you through the procedure for creating a custom profile, from which a custom color space is made. (The custom profile is stored at `~/Library/ColorSync/Profiles`.)

Figure 1   The Color pane of the Displays system preference
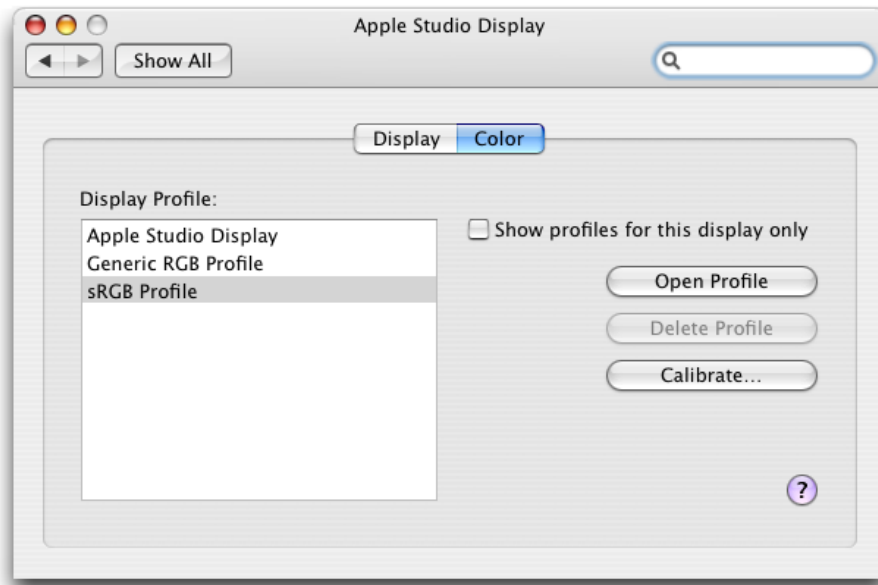
But if for some reason you need to create a custom color space programatically, you can use one of the initializer methods of the NSColorSpace class. You initialize a custom NSColorSpace object with one of two sources of data:

■   A ColorSync profile

■   An ICC profile

The NSColorSpace initializer `initWithICCProfileData:` method takes an NSData object encapsulating an ICC profile map. This profile map is a data structure depicted in Figure 2. The header of the profile map provides information applications need to search and sort ICC profiles, such as profile size, version, CMM type, color space, and primary platform. The tag table that follows consists of a variable number of entries; each entry has a unique tag signature, an offset to the beginning of the tag element data, and the size of that data. The International Color Consortium (ICC) maintains a website at http://www.color.org/icc_specs2.html from which you can obtain the most recent ICC profile specification.

Figure 2   An ICC profile map
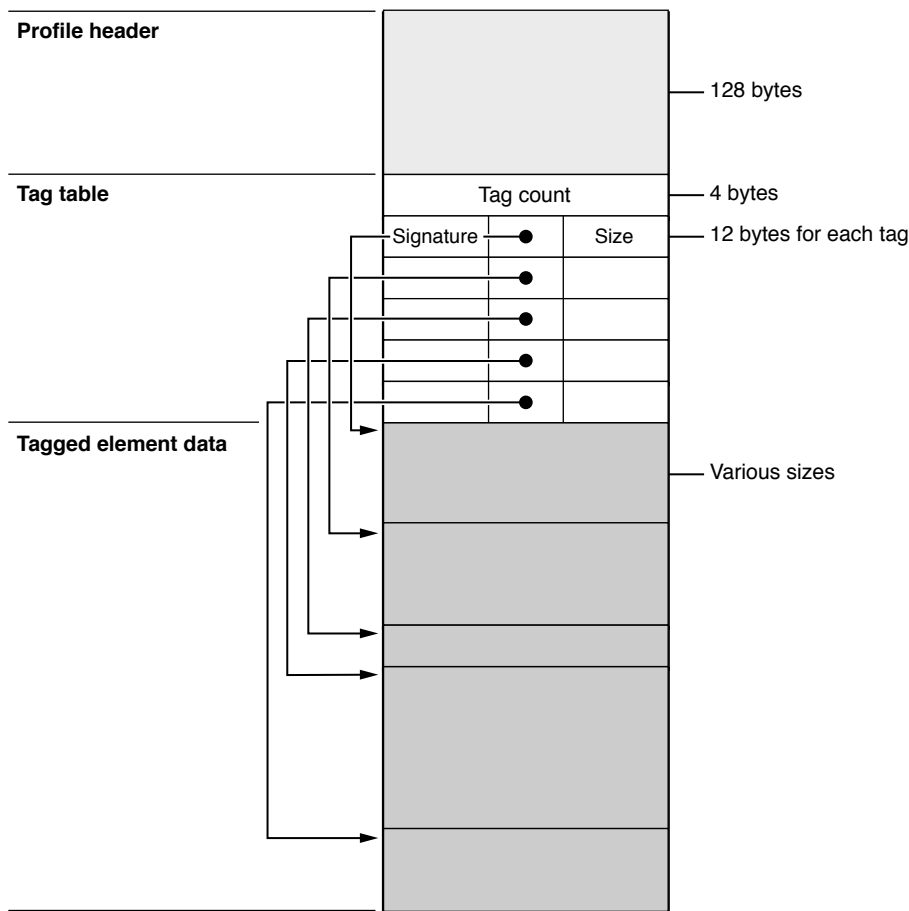
Once you have the ICC profile map constructed you can initialize an NSData object with it using the class factory methods `dataWithBytes:length:` or `dataWithBytesNoCopy:length:freeWhenDone:`. Then you can invoke the NSColorSpace initializer `initWithICCProfileData:`, passing in the data object.

The other alternative for creating an object representing a custom color space is to initialize the object with a `CMProfileRef` "object." A `CMProfileRef` opaque type represents a ColorSync profile. Typically you call the `CMNewProfile` function to create a new (but incomplete) profile and backing copy in a specific location. The function takes a pointer to the location and returns a pointer to the created `CMProfileRef`.

```
CMError CMNewProfile (
    CMProfileRef * prof,
    const CMProfileLocation * theProfile
)
```

An ColorSync profile is identical to an ICC profile. You must fill in the profile header fields and populate the profile with tags and their element data. Then call the function `CMUpdateProfile` to save the element data to the profile file. Note that the default ColorSync profile contents include a profile header of type `CM2Header` and an element table. See *ColorSync Manager Reference* for details on how to supply the required ColorSync profile data.

You can view the data of existing ColorSync profiles by running the ColorSync utility. Launch this utility by clicking the Open Profile button of the Color pane of the Displays system preference. Figure 3 shows part of the ColorSync profile for the Apple Studio Display.

Figure 3   ColorSync Utility showing values of ICC map
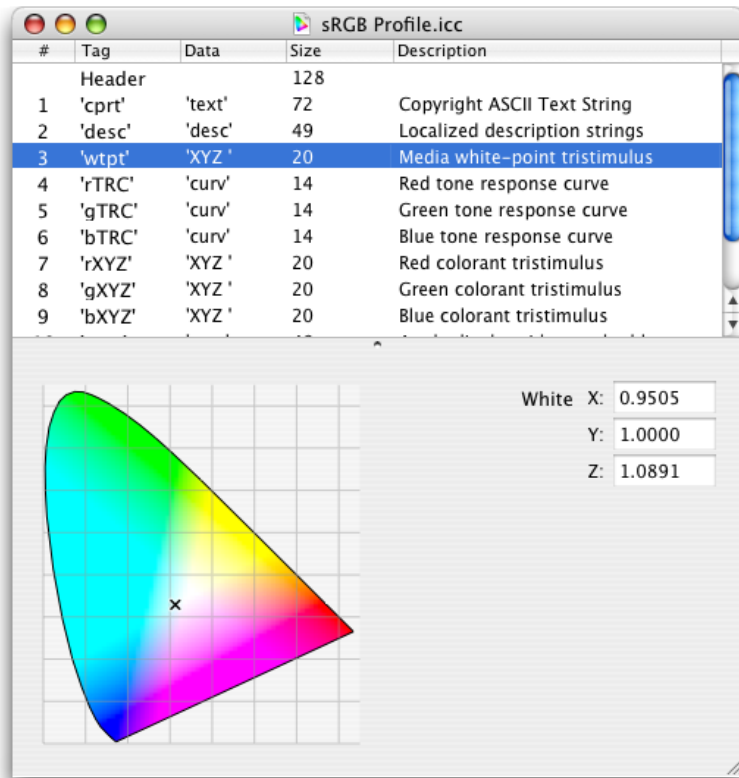
Accessing a Color's Components

There's usually no need to retrieve the individual components of a color, but when needed, you can retrieve either a set of components (using such methods as `getRed:green:blue:alpha:`) or an individual component (using such methods as `redComponent`). However, it's illegal to ask an NSColor for components that aren't defined for its color space. You can identify the color space by sending a `colorSpaceName` message to the NSColor object. If you need to ask an NSColor for components that aren't in its color space (for instance, when you've gotten the color from the color panel), first convert the color to the appropriate color space using the `colorUsingColorSpaceName:` method. If the color is already in the specified color space, you get the same color back; otherwise you get a conversion that's usually lossy or that's correct only for the current device. You get back `nil` if the specified conversion can't be done.

Accessing System Colors

`NSColor` has a number of methods that return system colors: colors controlled by user preferences. These colors—currently only `selectedControlColor` and `selectedTextBackgroundColor`—should be used by developers who want to create custom controls or subclass existing controls while honoring the user's color preferences.

> **Note:** A number of system colors, while still valid, are no longer meaningful under Aqua. These include any of the ones with "control" in their name. They return the old Platinum appearance colors which are usually a shade of gray.

System colors are implemented as named colors in a special color list named "`Developer`." You can examine this color list in the color panel of any application that supports colors. For more on named colors and color lists, see "About Color Lists" (page 17).

To extract the components of a system color, you must use the `NSColor` method `colorUsingColorSpaceName:` to convert the color to a color space known to respond to the component accessor methods you need; see "Creating and Converting Colors Using Color Spaces" (page 19) for more about color conversion.

An `NSSystemColorsDidChangeNotification` is sent when the system colors have been changed (such as through a system control panel interface). If you have any non-system colors that depend on the system colors, you can change them when you receive this notification.

Using the System Control Tint

Mac OS X allows a user to set the color used in the display of windows, menus and controls using the Appearance pane in System Preferences. This color is referred to as the control tint. User interface elements provided by the Application Kit automatically modify their appearance based on the current control tint.

Custom controls and other aspects of your application's user interface can also use the control tint as shown in Figure 1.

Figure 1   Examples of control tint aware custom control cells

Custom Control using Blue Tint   Custom Control using Graphite Tint

Getting the System Control Tint and Color

You can get the current system-wide control tint using the NSColor class method `currentControlTint`. This method returns an `NSControlTint` that represents the currently selected appearance color. Currently `NSBlueControlTint` or `NSGraphiteControlTint` are the possible return values representing the Aqua and Graphite appearances respectively.

Having determined the current `NSControlTint` value, you can now get the corresponding color using the `NSColor` class method `colorForControlTint:`.

An application can detect when the user changes the system-wide appearance setting, by registering as an observer of the `NSControlTintDidChangeNotification` notification.

Interaction with Views and Images

Your application may want to modify the appearance of the contents of a NSView subclass, or change the type of image used by an NSControl subclass in response to the system-wide appearance being changed. To do this your application must register for the `NSControlTintDidChangeNotification`, update the appropriate elements, and redraw the view.

The example in Listing 1 demonstrates how to determine the current control tint and change the image displayed in an NSButton and an NSImageView. The object that implements this method has previously been registered as an observer of the `NSControlTintDidChangeNotification` notification, with this method as the selector.

Listing 1   Example that sets the contents of an NSButton and NSImageView using the currentControlTint

```
- (void)systemTintChangedNotification:(NSNotification *)notification;
{
    NSString *tintImageName;

    // compare the result of [NSColor currentControlTint]
    // with the supported tint values, defaulting to Aqua
    if ([NSColor currentControlTint] == NSGraphiteControlTint)
        tintImageName=@"GraphiteImage";
    else
        tintImageName=@"AquaImage";

    [exampleButton setImage:[NSImage imageNamed:tintImageName]];
    [exampleImageView setImage:[NSImage imageNamed:tintImageName]];

}
```

Interaction with NSCell Subclasses

When the system-wide appearance is changed, any NSCell objects are automatically redrawn. It is the responsibility of an NSCell subclass to determine the current control tint as part of its implementation of `drawWithFrame:inView:`.

The example in Listing 2 demonstrates how to implement a tint-aware `drawWithFrame:inView:` method. It determines if the cell is using the system-wide appearance tint, or if its tint has been set explicitly and the appropriate image is then selected for display. The appropriate control tint color is also determined for drawing the clock's hands.

Listing 2   Example of a tint-aware NSCell drawWithFrame:inView: implementation

```
- (void)drawWithFrame:(NSRect)cellFrame
             inView:(NSView *)controlView
{
    // if we're not the front window, we'll resort to using the
    // special NSClearControlTint value
    NSControlTint currentTint;
    if ([[controlView window] isKeyWindow])
        currentTint = [self controlTint];
    else
        currentTint= NSClearControlTint;

    // If the NSCell's control tint has been overridden
    // using the setControlTint: method we should use
    // the value returned by [self controlTint] as this
    // controls authoritative tint. If the tint is
    // NSDefaultControlTint then this cell should use the
    // system-wise appearance value, and we use the value
    // returned by the NSColor +currentControlTint method.
```

```
    if ([self controlTint] == NSDefaultControlTint)
        currentTint=[NSColor currentControlTint];
    else
        currentTint=[self controlTint];



    // and change the image used in drawing according to
    // the currentTint
    // Use the Aqua image as the default image
    NSImage *clockFaceImage;
    switch (currentTint) {
        case NSGraphiteControlTint:
            clockFaceImage = [NSImage imageNamed: @"ClockFace-Graphite"];
            break;
        case NSClearControlTint:
            clockFaceImage = [NSImage imageNamed: @"ClockFace-Clear"];
            break;
        case NSBlueControlTint:
        default:
            clockFaceImage = [NSImage imageNamed: @"ClockFace-Aqua"];
            break;
    }

    float clockRadius = MIN(NSHeight(cellFrame), NSWidth(cellFrame));

    // Draw the clock face (draw it flipped
    // if we are in a flipped view, like NSMatrix).
    [clockFaceImage setFlipped:[controlView isFlipped]];
    [clockFaceImage drawInRect:NSMakeRect(NSMinX(cellFrame),
                                          NSMinY(cellFrame),
                                          clockRadius,clockRadius)
                    fromRect:NSMakeRect(0,0,
                                          [clockFaceImage size].width,
                                          [clockFaceImage size].height)
                    operation:NSCompositeSourceOver
                     fraction:1.0];

    // get the color for the currentTint and use it for
    // drawing the hands on the clock face
    NSColor *tintColor=[NSColor colorForControlTint:currentTint];

    // Draw the clock hour and minute hands.
    [self drawClockHandsForTime:time
                    withFrame:cellFrame
                       inView:controlView
                    usingColor:tintColor];
}
```

Choosing Colors With Color Wells and Color Panels

A color well displays and lets the user select a single color value. A user can set a color well's value by dragging a color to it or by clicking the color well and using the color panel that appears. A color panel contains several color pickers that let the user select a specific color. You can choose which color pickers are displayed and add new ones. See "Choosing the Color Pickers in a Color Panel" (page 35) and "Adding Custom Color Pickers to a Color Panel" (page 37).

Using Color Wells

NSColorWell is an NSControl for selecting and displaying a single color value. An example of an NSColorWell object (or simply color well) is found in NSColorPanel, which uses a color well to display the current color selection. A color well is available from the Palettes panel of Interface Builder.

An application can have one or more active color wells. You can activate multiple color wells by invoking the `activate:` method with `NO` as its argument. When a mouse-down event occurs on a color well's border, it becomes the only active color well. When a color well becomes active, it brings up the color panel also.

The `mouseDown:` method enables a color well to send its color to another color well or any other subclass of NSView that implements the NSDraggingDestination protocol.

Using Color Panels

NSColorPanel provides a standard user interface for selecting color in an application. It provides a number of standard color selection modes, and, with the NSColorPickingDefault and NSColorPickingCustom protocols, allows an application to add its own color selection modes. It allows the user to save swatches containing frequently used colors. Once set, these swatches are displayed by NSColorPanel in any application where it is used, giving the user color consistency between applications. NSColorPanel enables users to capture a color anywhere on the screen for use in the active application, or to drag a color from the color panel into an application view.

When you select a color in the panel, NSColorPanel sends a `changeColor:` message to the first responder. It also sends its action message (set by `setAction:`) to its target object (set by `setTarget:`), provided that neither the action nor the target is `nil`. NSColorPanel also sends its action to its target whenever you select a color in the color panel.

An application has only one instance of NSColorPanel, the shared instance. Invoking the `sharedColorPanel` method returns the shared instance of NSColorPanel, instantiating it if necessary.

You can put NSColorPanel in any application created with Interface Builder by adding the "Colors…" item from the Menu palette to the application's menu.

The NSColorList class provides an API for managing custom color lists. The NSColorPanel methods `attachColorList:` and `detachColorList:` let your application add and remove custom lists from the NSColorPanel object's user interface. For more information, see "About Color Lists" (page 17).

NSColorPanel dynamically loads NSColorPicker objects from the following directories:

```
~/Library/ColorPickers/
/Local/Library/ColorPickers/
/System/Library/ColorPickers/
```

Choosing the Color Pickers in a Color Panel

The color mask determines which of the color modes are enabled for an `NSColorPanel` object. This mask is set before you initialize a new instance of `NSColorPanel`. `NSColorPanelAllModesMask` represents the logical OR of the other color mask constants: It causes the `NSColorPanel` object to display all standard color pickers. When initializing a new instance of `NSColorPanel`, you can logically OR any combination of color mask constants to restrict the available color modes.

Mode	Color Mask Constant
Grayscale-Alpha	`NSColorPanelGrayModeMask`
Red-Green-Blue	`NSColorPanelRGBModeMask`
Cyan-Yellow-Magenta-Black	`NSColorPanelCMYKModeMask`
Hue-Saturation-Brightness	`NSColorPanelHSBModeMask`
Custom palette	`NSColorPanelCustomPaletteModeMask`
Custom color list	`NSColorPanelColorListModeMask`
Color wheel	`NSColorPanelWheelModeMask`
All of the above	`NSColorPanelAllModesMask`

The a color panel's color mode mask is set using the class method `setPickerMask:`. The mask must be set before creating an application's instance of `NSColorPanel`.

When an application's instance of `NSColorPanel` is masked for more than one color mode, your program can set its active mode by invoking the `setMode:` method with a color mode constant as its argument; the user can set the mode by clicking buttons on the panel. Here are the standard color modes and mode constants:

Mode	Color Mode Constant
Grayscale-Alpha	`NSGrayModeColorPanel`
Red-Green-Blue	`NSRGBModeColorPanel`
Cyan-Yellow-Magenta-Black	`NSCMYKModeColorPanel`
Hue-Saturation-Brightness	`NSHSBModeColorPanel`
Custom palette	`NSCustomPaletteModeColorPanel`
Custom color list	`NSColorListModeColorPanel`

Mode	Color Mode Constant
Color wheel	`NSWheelModeColorPanel`

In grayscale-alpha, red-green-blue, cyan-magenta-yellow-black, and hue-saturation-brightness modes, the user adjusts colors by manipulating sliders. In the custom palette mode, the user can load an NSImage file (TIFF or EPS) into the color panel, then select colors from the image. In custom color list mode, the user can create and load lists of named colors. The two custom modes provide pop-up buttons for loading and saving files. Finally, color wheel mode provides a simplified control for selecting colors.

If a color panel has been used, it uses whatever mode it was in last as the default mode when `NSColorPanelAllModesMask` is used to initialize the `NSColorPanel`. Otherwise, it uses color wheel mode.

Adding Custom Color Pickers to a Color Panel

A color picker is a user interface for color selection in an `NSColorPanel` object. The color panel lets the user select a color picker from a matrix of `NSButtonCell` objects across the top of the panel. A color picker is a loadable bundle with an extension of `.colorPicker` that can be installed in one of four places:

- `/System/Library/ColorPickers` — Apple-supplied color pickers only

- `/Library/ColorPickers`

- `~/Library/ColorPickers`

- Inside the application bundle in a subdirectory of the `Resources` directory named `ColorPickers`, for example:

  `TextEdit.app/Contents/Resources/ColorPickers/MyColorPickers.colorPicker`

The bundle should contain all required resources for the color picker, including nib files, image files, and so on. These resources should be internationalized for each supported localization. `NSColorPanel` allocates and initializes an instance of each class for each color-picker bundle found in these locations. The class name is assumed to be the bundle directory name minus the `.colorPicker` extension. If you have an icon in TIFF format (including a `.tiff` extension) and with the same name as that of the color-picker class, the color panel displays it in the button cell at the top of the panel.

Overview of the Color Picker API

The `NSColorPickingDefault` and `NSColorPickingCustom` protocols provide an interface for adding custom color pickers to an application's color panel. The `NSColorPickingDefault` protocol provides basic behavior for a color picker. The `NSColorPickingCustom` protocol provides implementation-specific behavior.

The `NSColorPicker` class implements the `NSColorPickingDefault` protocol. To implement your own color picker you must create a subclass of `NSColorPicker` and implement the `NSColorPickingCustom` protocol for that subclass. You can also re-implement any `NSColorPickingDefault` methods if there is a need to; for example, you could write code to supply a custom tool tip or an image with a name other than the color-picker class name. You must also implement a view containing the actual color-selection user interface to be inserted into the color panel for the color picker. The custom `NSColorPicker` object should have an outlet connecting it to this view.

The color-picker API requires that you specify supported color-picker modes. For a list of the existing color picker modes and mode constants, see "Choosing the Color Pickers in a Color Panel" (page 35). If your color picker includes submodes, you should define a unique value for each submode. As an example, the slider picker has four values defined in the above list (`NSGrayModeColorPanel`, `NSRGBModeColorPanel`, `NSCMYKModeColorPanel`, and `NSHSBModeColorPanel`)—one for each of its submodes.

# The Procedure for Creating a Custom Color Picker

What follows is a short tutorial for creating custom color pickers that are automatically added to a color panel. The procedure is fairly easy, but there are a few details you should be aware of. The tutorial makes use of the `RadiantPicker` example project installed in `/Developer/Examples/AppKit`.

> **Note:** To ensure the accuracy of the colors in your custom color picker, you should handle color spaces correctly. See "Working With Color Spaces" (page 19) (especially the section on programming guidelines) for information on the proper use of color spaces.

Start by creating a project for a Cocoa bundle: Choose New Project from the File menu and then select Cocoa Bundle from the list of project types. For the name of the bundle enter the same name you intend to give to the name of your `NSColorPicker` subclass.

The next step is not absolutely necessary, but is helpful for debugging and testing the custom color picker. As shown in Figure 1, double-click the project folder in Xcode and, in the General pane of the project preferences, set the build product output directory to `Library/ColorPickers` in your home directory.

Figure 1   Specifying the location for the built color picker
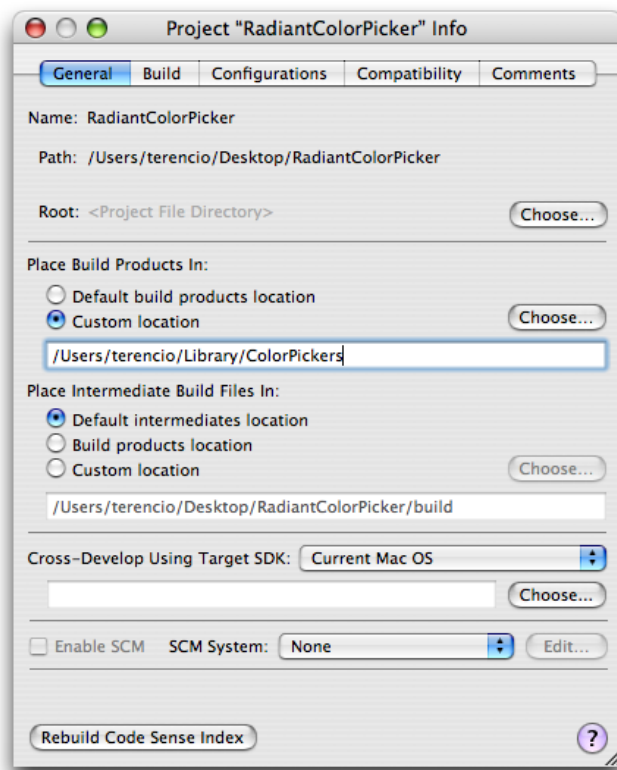
> **Note:** For this option to be effective, you must turn off an Xcode user default. Specifically, enter the following at the command line:
>
> ```
> defaults write com.apple.xcode UsePerConfigurationBuildLocations NO
> ```

Next set the extension of the color-picker bundle: Double-click the bundle target, select the Build pane, and search for the Wrapper Extension setting. Change the value of this setting to "colorPicker", as shown in Figure 2.

Figure 2   Setting the bundle extension
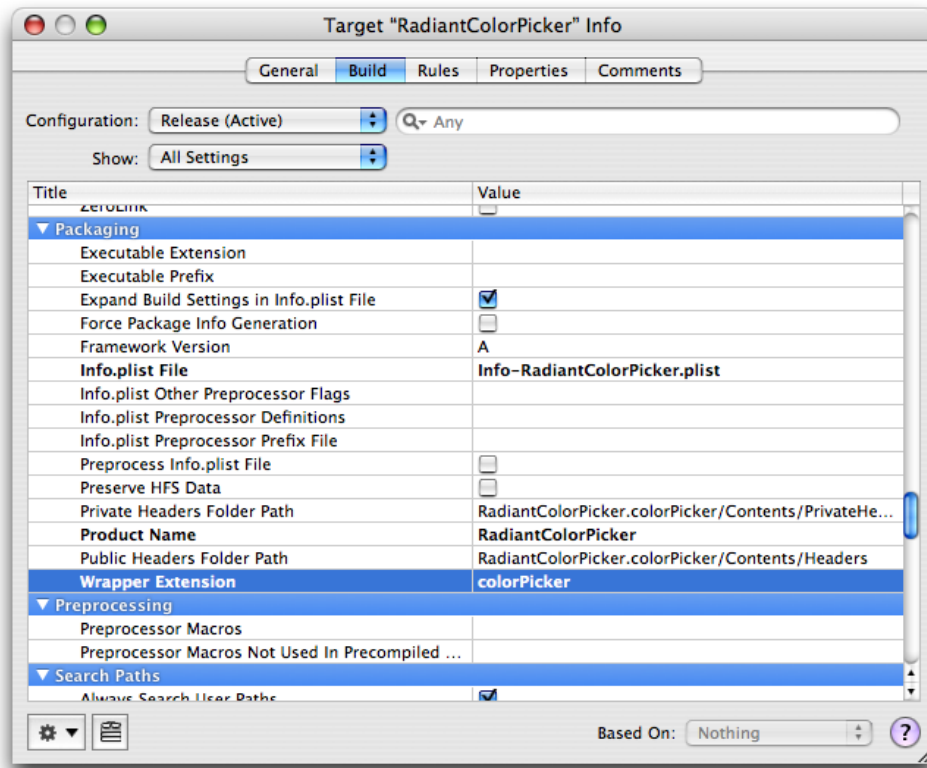
The final step in project configuration is to set two properties: the bundle identifier and the principal class. The name of the principal class should be the same as the name of the project. Figure 3 illustrates what these settings might look like.

Figure 3 Setting bundle identifier and principal class
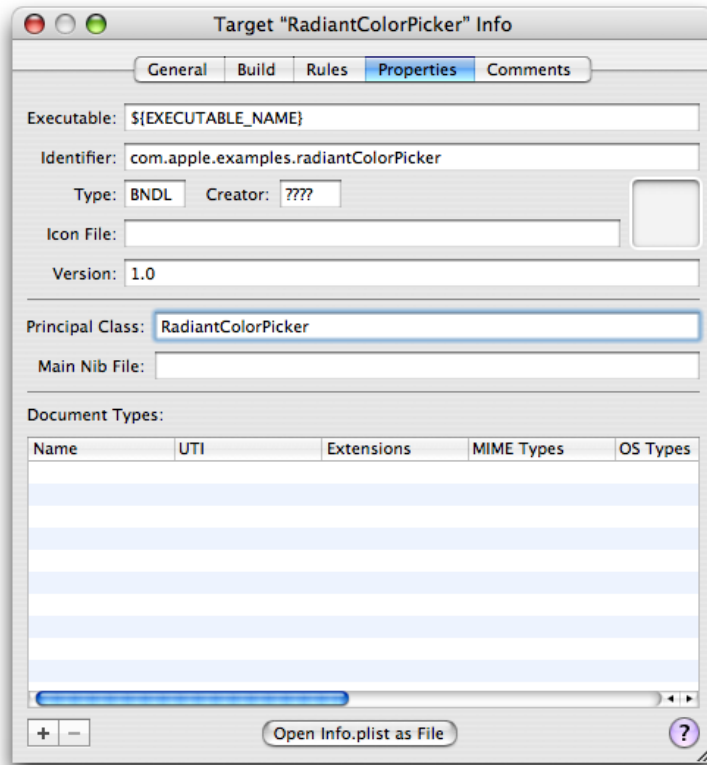
Now let's leave Xcode for awhile and switch to Interface Builder. The Cocoa Bundle project type does not include a nib file, so you'll have to add one. Launch the application and in the Starting Point panel select Empty under Cocoa and click New (see Figure 4).

Figure 4 Creating a nib file for the color picker
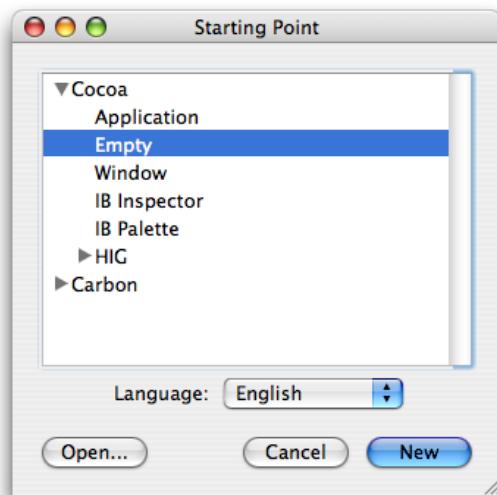
Drag a Custom View object from the Containers palette onto your nib file window (this step requires Mac OS X v10.4 or greater). Then construct the user interface of your color picker on this view with palette objects (if necessary). Figure 5 gives an example of what you might see in Interface Builder. Size the view so that it is commensurate with the `NSSize` value returned from the `NSColorPicker` implementation of the `NSColorPickingDefault` protocol method `minContentSize`. Save the nib file under an appropriate name and then, in Xcode, add the nib file to the project as a localized resource.

Figure 5 The nib file window for the color picker

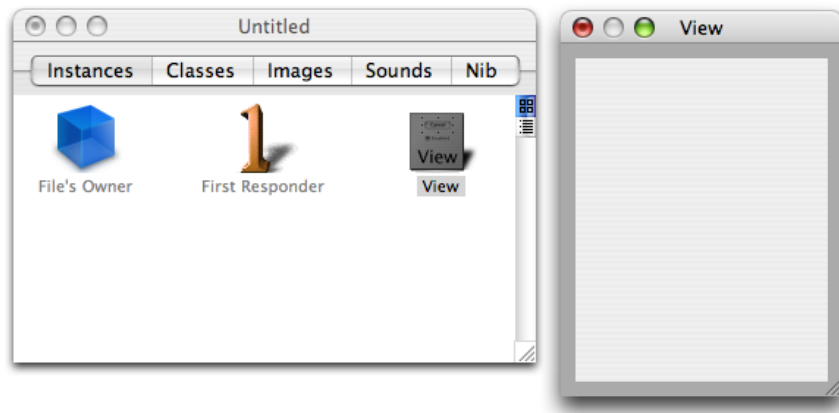Now comes time for some coding, starting with your custom subclass of `NSColorPicker`. In Xcode, create the files for a new Objective-C class and add them to the project (File > New File). Give the header and implementation files the same name (minus the extensions) as that of the project. Open the header file and change the superclass to `NSColorPicker` and declare that the class adopts the `NSColorPickingCustom` protocol. Also add any outlets you require to the view containing the color picker's user interface. Listing 1 illustrates what the header file declarations should look like.

Listing 1 Declarations of the `NSColorPicker` subclass

```
@interface RadiantColorPicker: NSColorPicker <NSColorPickingCustom> {
    IBOutlet NSView *_pickerView;
    IBOutlet RadiantColorControl *_radiantColorControl;
}
@end
```

To keep Interface Builder synchronized with these changes, drag the header file from Xcode and drop in on the nib file window. Select the File's Owner icon in the nib file window, open the Custom Class inspector pane (Command-5), and select the `NSColorPicker` subclass you just declared. Then connect any outlets you defined for that class.

At this point, create and add files for any custom classes needed for the user interface of the color picker. As with the custom `NSColorPicker` subclass, added any required declarations to the header file and then drag this file onto the nib file window to make Interface Builder aware of the custom view. Assign the custom view class using the Custom Class inspector pane. You may need to do this before making any outlet connections between the custom view class and your `NSColorPicker` subclass.

Implement your `NSColorPicker` subclass by, at the least, implementing the methods of the `NSColorPickingCustom` protocol and, optionally, re-implementing methods of the `NSColorPickingDefault` protocol, as necessary. Listing 2 provides an example.

Listing 2   Sample implementation of the custom NSColorPicker class

```
@implementation RadiantColorPicker

- (id)initWithPickerMask:(NSUInteger)mask colorPanel:(NSColorPanel
*)owningColorPanel {
    return [super initWithPickerMask:mask colorPanel:owningColorPanel];
}

- (void)dealloc {
    [_pickerView release];
    [super dealloc];
}

- (BOOL)supportsMode:(NSColorPanelMode)mode {
    return (mode == NSRGBModeColorPanel) ? YES : NO;
}

- (NSColorPanelMode)currentMode {
    return NSRGBModeColorPanel;
}

- (NSView *)provideNewView:(BOOL)initialRequest {
    if (initialRequest) {
        // Load our nib files
        if (![NSBundle loadNibNamed:@"RadiantColorPicker" owner:self]) {
            NSLog(@"ERROR: couldn't load MyColorPicker nib");
        }
        [_radiantColorControl setTarget:self];
        [_radiantColorControl setAction:@selector(colorChanged:)];
    }
    return _pickerView;
}

- (void)colorChanged:(id)sender {
    [[self colorPanel] setColor:[_radiantColorControl color]];
}

- (void)setColor:(NSColor *)newColor {
    [_radiantColorControl setColor:newColor];
}

- (NSString *)buttonToolTip {
    return NSLocalizedString(@"Radiant Picker", @"Tooltip for the radiant color
 picker button in the color panel");
}

@end
```

There are a few things to note about this code:

■ The class implements every method of the NSColorPickingCustom protocol.

■ It reimplements one method of NSColorPickingDefault, buttonToolTip; from this implementation it returns a localized string (meaning that it would probably have a Localizable.strings file for each supported localization).

> **Important:** The `buttonToolTip` method, along with the `minContentSize` method, were added to the interface of `NSColorPickingDefault` for Mac OS X v10.5.

■ As shown in the example, do *not* load your color picker's nib file in the initializer `initWithPickerMask:colorPanel:`. Instead load the nib file in the `provideNewView:` method. You only need to load the nib file if the sole argument of the `provideNewView:` method is `YES`; otherwise, return the cached instance.

■ Two of the methods manage the color-panel modes supported by the color picker (in this case, only `NSRGBModeColorPanel`). If the picker masks of the control panel do not contain a supported mode, the color picker is not asked to load itself into the color panel.

■ The custom view of this color picker—which is an instance of an `NSControl` subclass—sends a (private) `colorChanged:` action message to the custom `NSColorPicker` object upon each mouse-up event. In this method, the color picker sets the current color of its color panel. Note the code in `provideNewView:` that sets the target and action of the custom control object.

■ Because it's File's Owner, the custom `NSColorPicker` class is responsible for releasing top-level nib objects such as the color picker's custom view (`_pickerView`).

Implement the custom view objects for your color picker as necessary.

When you have finished implementing and testing your custom color picker, and it is a third-party product, create an installer for it that puts in one of the prescribed locations.

Subclassing NSColor

Subclasses of NSColor need to implement the `colorSpaceName` and `set` methods, as well as the methods that return the components for that color space and the methods in the NSCoding protocol. Some other methods—such as `colorWithAlphaComponent:`, `isEqual:`, and `colorUsingColorSpaceName:device:`—may also be implemented if they make sense for the color space. Mutable subclasses (if any) should additionally implement `copyWithZone:` to provide a true copy.

Storing NSColor in User Defaults

It is often desirable to store the value of an NSColor instance in an application's user defaults. However, NSUserDefaults only supports the storage of objects that can be represented in an property list.

The solution is to use object archiving to write the NSColor instance data to an NSData instance and then store that as the default as shown in Listing 1. This is often done in an application life-cycle exit point such as the `applicationShouldTerminate:` delegation method.

Listing 1   Storing an NSColor instance in user defaults

```
// store the value in aColor in user defaults
// as the value for key aKey
NSData *theData=[NSArchiver archivedDataWithRootObject:aColor];
[[NSUserDefaults standardUserDefaults] setObject:theData forKey:aKey];
```

To read the value back from NSUserDefaults an application retrieves the NSData instance for the required key and unarchives the NSColor instance. The example in Listing 2 demonstrates retrieving the color. This is often done in an application life-cycle entry point such as `awakeFromNib`.

Listing 2   Retrieving an NSColor instance from user defaults

```
// read the value of the user default with key aKey
// and return it in aColor
NSColor * aColor =nil;
NSData *theData=[[NSUserDefaults standardUserDefaults] dataForKey:aKey];
if (theData != nil)
    aColor =(NSColor *)[NSUnarchiver unarchiveObjectWithData:theData];
```

Extending NSUserDefaults to support NSColor

It's possible to take advantage of the support for categories in Objective-C to add NSColor support to the existing NSUserDefaults class, without subclassing.

The example code in Listing 3 and Listing 4 shows an implementation of such a category. The method `setColor:forKey:` in archives the specified color to an NSData instance and stores it in the user defaults using the specified key. The method `colorForKey:` retrieves the NSData instance specified by the key, and then unarchives an instance of NSColor using the data.

Listing 3   Contents of NSUserDefaults myColorSupport category .h file

```
#import <Foundation/Foundation.h>

@interface NSUserDefaults(myColorSupport)
- (void)setColor:(NSColor *)aColor forKey:(NSString *)aKey;
- (NSColor *)colorForKey:(NSString *)aKey;
@end
```

Listing 4   Contents of NSUserDefaults myColorSupport category .m file

```
#import "NSUserDefaults+myColorSupport.h"

@implementation NSUserDefaults(myColorSupport)

- (void)setColor:(NSColor *)aColor forKey:(NSString *)aKey
{
    NSData *theData=[NSArchiver archivedDataWithRootObject:aColor];
    [self setObject:theData forKey:aKey];
}

- (NSColor *)colorForKey:(NSString *)aKey
{
    NSColor *theColor=nil;
    NSData *theData=[self dataForKey:aKey];
    if (theData != nil)
        theColor=(NSColor *)[NSUnarchiver unarchiveObjectWithData:theData];
    return theColor;
}

@end
```

> **Important:** There is some risk in implementing a category with method names that are common enough that Apple could use them in the future. An alternative would be to use prefixes that Apple would not use, for example, `my_colorForKey:`.

Establishing Bindings Between Colors and User Defaults

You can easily establish a binding between a user-interface object whose value is a color (that is, an `NSColor` object) and user defaults. When the user chooses a color preference for something in an application, the binding preserves and restores the preference across successive launches of the application.

To effect the binding, use a ready-made instance of the `NSUnarchiveFromDataTransformerName` value transformer in Interface Builder. An `NSValueTransformer` object converts an object value typically in two directions: between the form in which it is displayed and the form in which it is stored. The `NSUnarchiveFromDataTransformerName` value transformer works by archiving an `NSColor` object in an `NSData` object and then, on the other side of the binding, unarchiving the color object from the data object. For this value transformation to work, the archived object must implement the `NSCoding` protocol using sequential archiving—which `NSColor` does.

An `NSColorWell` instance is a user-interface object whose value is a `NSColor` object. You can drag the color-well object from the Controls palette of Interface Builder onto a view. To establish the binding between this object and user defaults, complete the following steps:

1. With the color well still selected, open the Bindings pane of the Inspector and expose the **value** binding.

2. From the "Bind to" pop-up menu choose Shared User Defaults.

   This action adds an instance of `NSUserDefaultsController` ("Shared Defaults") to the nib file window.

3. Keep the Controller Key field as `values` but in the Model Key Path field specify a name under which to save the color object (`theColor`, in this example).

4. From the Value Transformer combo box select (or enter) `NSUnarchiveFromData`.

When you're finished, your setup in Interface Builder should look similar to that in Figure 1.

Figure 1 Establishing a binding between an `NSColor` value and user defaults
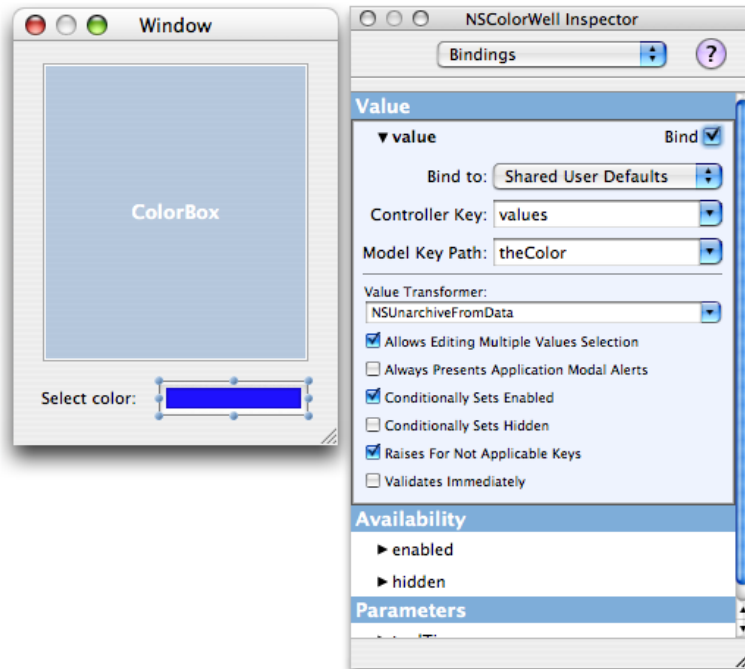
If at this point you save your nib file and build your project, you can launch the application, change the color in the color well, quit the application, and then relaunch. The color in the color well is what it was when you last changed it.

Although the foregoing procedure establishes a binding between an `NSColor` value of a view and user defaults, it does not propagate changes in that value to other objects in the application. You can do that by explicitly setting the color to the restored default when the application launches and, thereafter, by having the first responder handle the `changeColor:` message whenever the user changes the color. But you can also use bindings so that any change in color value is propagated both to user defaults and applied to a custom view in the application. This requires you to complete the following steps:

- Declare an `NSColor` property of the custom view class.

- Expose this property as a binding (`exposeBinding:`); do this in the class method `initialize`.

- In the setter method for the property, send `setNeedsDisplay:` (or `setNeedsDisplayInRect:`) to `self` after the new color is retained; this forces the view to redraw itself in the new color.

- Define a controller object that acts as application delegate. When the application finishes launching, this object establishes a binding between the custom view's `NSColor` property and the property of the `NSUserDefaultsController` object bound to the color well.

 See Listing 5 for an example of this final step.

Listing 5   Establishing a binding between an `NSColor` property and `NSUserDefaultsController`

```
@implementation AppDelegate
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    [theColorBox bind:@"backgroundColor" toObject:[NSUserDefaultsController
sharedUserDefaultsController]
        withKeyPath:@"values.theColor"
        options:[NSDictionary dictionaryWithObject:NSUnarchiveFromDataTransformerName
forKey:NSValueTransformerNameBindingOption]];
}
@end
```

Document Revision History

This table describes the changes to *Color Programming Topics for Cocoa*.

Date	Notes
2009-03-04	Removed statement that drawing to printers doesn't support alpha.
2008-10-15	Added cross-reference in "Adding Custom Color Pickers to a Color Panel" to "Working WIth Color Spaces."
2006-10-03	Clarified named color spaces and system colors, gave an example of binding NSColor to user defaults, and showed how to make a custom color picker.
2005-04-29	Added "Working WIth Color Spaces" and rewrote "About Color Spaces." Changed title of document from "Using Color."
2003-10-11	Added the task "Using the System Control Tint" (page 29).
2003-02-17	Added the task "Storing NSColor in User Defaults" (page 47).
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.