Freescale Semiconductor
Application Note

Document Number: AN4042
Rev. 0, 05/2010

# Image Processing API for the i.MX Platform

*by Multimedia Applications Division*
*Freescale Semiconductor, Inc.*
*Austin, TX*

This application note provides information about the image processing API for the i.MX platform. Image processing is a form of signal processing. The input for image processing is an image, such as a photograph or frame of video. The output can be an image or a set of characteristics or parameters related to the image. Most of the image processing techniques treat the image as a two-dimensional signal and applies the standard signal processing techniques to it.

Image processing usually refers to digital image processing. However, optical and analog image processing are also possible. The application note describes the general techniques that apply to all types of image processing. The source code is in ANSI C that makes the file portable between any Operating System (OS).

Contents

1. 24-Bit RGB Representation . . . . . . . . . . . . . . . . . . . . 2
2. Raw Images . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
3. Frame Buffer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
4. Writing the Frame Buffer . . . . . . . . . . . . . . . . . . . . . . 4
5. Geometric Transformation . . . . . . . . . . . . . . . . . . . . . 5
6. Color Transformation . . . . . . . . . . . . . . . . . . . . . . . . 10
7. Drawing on Screen in Windows® CE . . . . . . . . . . . . 15
8. Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
9. Revision History . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

© 2010 Freescale Semiconductor, Inc. All rights reserved.

freescale™ semiconductor

# 1 24-Bit RGB Representation

The RGB values encoded in 24 bits per pixel (bpp) are specified using three 8-bit unsigned integers (0–255) that represent the intensities of red, green, and blue. This representation is the current mainstream standard representation for the true color and common color interchange in image file formats, such as JPEG or TIFF. The 24-bit RGB representation allows more than 16 million combinations. Therefore, some system uses the term millions of colors for this mode. Most of these colors are visible to the human eye.

Figure 1 shows three fully saturated faces of a 24 bpp RGB cube, unfolded into a plane.

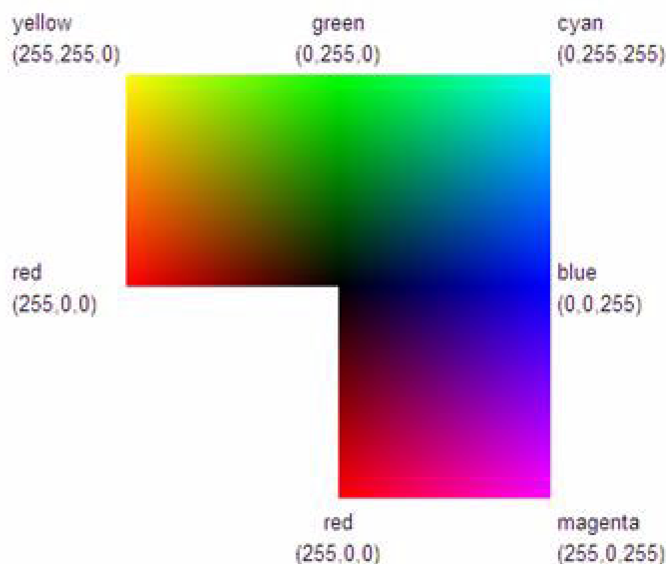


**Figure 1. 24 bpp RGB Cube**

# 2 Raw Images

A raw image file contains minimally processed data from the image sensor of a digital camera or an image scanner. These files are termed as raw files because they are not processed and are ready to be printed or used with a bitmap graphics editor. In general, an image is processed by a raw converter in a wide gamut internal color space. Here, precise adjustments are made before converting the raw images to an RGB file format, such as TIFF or JPEG for storage, printing or for further manipulation. These images are often described as RAW image files (note capitalization), based on the erroneous belief that they represent a single file format. All these files have `.RAW` as a common filename extension.

**NOTE**

Several raw image file formats are used in different models of digital cameras.

Raw image files are sometimes called digital negatives, as they fulfill the same role as film negatives in traditional chemical photography. The negative cannot be used as an image, but has all the information needed to create an image. Likewise, the process of converting a raw image file into a viewable format is

called developing a raw image. This is in analogy with the film development process, which is used to convert photographic film into viewable prints.

A raw image in C is represented as given below:

```
unsigned char raw_image[]={
0x00,0x3d,0x10,0x81,0x00,0x3d,0x10,0x81,0x45,
0x00,0x3d,0x10,0x81,0x00,0x3d,0x10,0x81,0x71,
0x00,0x3d,0x10,0x81,0x00,0x3d,0x10,0x81,0x22,
0x00,0x3d,0x10,0x81,0x00,0x3d,0x10,0x81,0x31
}
```

The above example can be interpreted as a $9 \times 4$ pixels image that uses 1 byte to represent the color palette. Image can also be of $3 \times 4$ pixels with a color depth of 3 bytes (16,777,216 colors).

# 3 Frame Buffer

A frame buffer is a video output device that drives a video display from a memory buffer that contains a complete frame of data. The information in the buffer consists of color values for every pixel (point that can be displayed) on the screen. Color values are commonly stored in 1-bit monochrome, 4-bit pelletized, 8-bit pelletized, 16-bit high color, and 24-bit true color formats. An additional alpha channel is sometimes used to retain the information about pixel transparency.

The total amount of memory required to drive the frame buffer depends on the resolution of the output signal, color depth, and palette size.

Equation 1 shows the formula to calculate the size of the frame buffer.

*frame buffer size (in bytes) = (pixels width) x (pixels height) x (number of colors per pixel)* **Eqn. 1**

A frame buffer object can be created as shown below:

```
FrameBufferProperties fbp;

fbp.color_depth= RGB_24BITS;
fbp.height= 100;
fbp.width= 100;

FrameBuffer fb = NewFrameBuffer(&fbp);
```

where `fbp` is an instance of the `FrameBufferProperties` that is used to set the color depth, height, and width of the frame buffer object. The `NewFrameBuffer` method allocates memory dynamically according to the parameters in the `FrameBufferProperties`. It returns a pointer to the allocated memory.

The `NewFrameBuffer` function is given as follows:

```
uchar8* NewFrameBuffer(FrameBufferProperties* fbp)
{
uint32 size= sizeof(uchar8)*(fbp->color_depth)*(fbp->height)*(fbp->width);
return (unsigned char*)malloc(size);
}
DeleteFrameBuffer is a macro that frees the allocated memory.
#define DeleteFrameBuffer(instance) free(instance)
```

# 4 Writing the Frame Buffer

As the `NewFrameBuffer` method does not initialize the allocated memory due to the `malloc` function, the `SetFrameBufferBackground` method is used to set a background color to the selected frame buffer.

The `SetFrameBufferBackground` takes the following format:

```
void SetFrameBufferBackground(FrameBuffer fb,FrameBufferProperties* fbp, uint32
background);
```

**Example 1. `SetFrameBufferBackground` function**

```
/*set fbp structure according to frame buffer properties*/
FrameBufferProperties fbp;

fbp.color_depth= RGB_24BITS;
fbp.height= 100;
fbp.width= 100;

/*set img structure according to raw image format*/
ImageProperties imgp;
imgp.color_depth= RGB_24BITS;
imgp.height= 16;
imgp.width= 16;

/*create two frame buffers*/
FrameBuffer fb = NewFrameBuffer(&fbp);
FrameBuffer ffb = NewFrameBuffer(&fbp);

/*set white color as background color*/
SetFrameBufferBackground(ffb,&fbp,0xffffff);

/*add a raw image to fb*/
/*consider that raw_img is already declared somewhere in code*/
AddRawImage(fb,&fbp,raw_img,&imgp,0,0);

/*HERE PASS THE ffb VARIABLE TO THE PLATFORM DEPENDENT DRAW METHOD OR DEVICE CONTEXT*/

/*release allocated memory*/
DeleteFrameBuffer(fb);
DeleteFrameBuffer(ffb);
```

The `AddRawImage` method copies a raw image on the selected frame buffer at the specified coordinates. It compares the size of the final frame buffer against the size of the raw image using the `isFrameBufferSuitable` method. This prevents the raw image from being copied completely.

The `AddRawImage` function takes the following format:

```
void AddRawImage(FrameBuffer fb, FrameBufferProperties* fbp, RawImage* img,
ImageProperties* imgp, int32 x, int32 y);
```

**Example 2. `AddRawImage` function**

```
/*set fbp structure according to frame buffer properties*/
FrameBufferProperties fbp;
fbp.color_depth= RGB_24BITS;
fbp.height= 100;
fbp.width= 100;
```

```
/*set img structure according to raw image format*/
ImageProperties imgp;
imgp.color_depth= RGB_24BITS;
imgp.height= 16;
imgp.width= 16;

/*create two frame buffers*/
FrameBuffer fb = NewFrameBuffer(&fbp);
FrameBuffer ffb = NewFrameBuffer(&fbp);

/*add a raw image to fb*/
/*consider that raw_img is already declared somewhere in code*/
AddRawImage(fb,&fbp,raw_img,&imgp,0,0);

/*HERE PASS THE ffb VARIABLE TO THE PLATFORM DEPENDENT DRAW METHOD OR DEVICE CONTEXT*/

/*release allocated memory*/
DeleteFrameBuffer(fb);
DeleteFrameBuffer(ffb);
```

# 5 Geometric Transformation

Transformations form the fundamental part of computer graphics. Transformations are used to position the objects, shape the objects, change the position of view, and change the perspective.

The four main types of transformations, which are performed in two-dimensions, are as follows:

- Translations
- Scaling
- Rotation
- Shearing

These basic transformations can be combined to form more complex transformations.

## 5.1 Translation

Translation with two-dimensional points is easy. The user only has to determine the distance that the object should be moved and add those variables to the previous x and y co-ordinates, respectively.

Figure 2 shows the original and translated image.

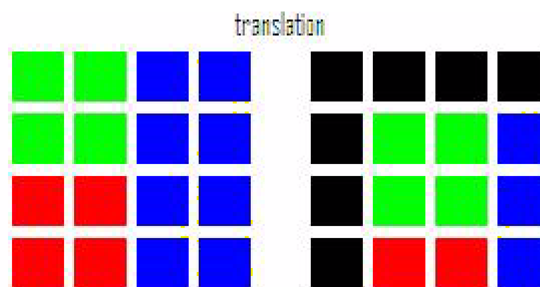


**Figure 2. Original and Translated Images**

The following gives an example for translation. Consider that the point (0,0) is on the upper left of the screen. If (0,0) has to be in the middle of the screen, the translation coordinates should be ([image height/2], [image width/2]).

The `Translate` function takes the following format:

```
void Translate(FrameBuffer tfb, FrameBuffer fb, FrameBufferProperties* fbp, int32 x,
int32 y);
```

where the five parameters of the method are as follows:

- Source frame buffer
- Final frame buffer
- Frame buffer properties
- Two-dimensional offset point in the x-axis
- Two-dimensional offset point in the y-axis

**Example 3. `Translate` function**

```
/*set fbp structure according to frame buffer properties*/
FrameBufferProperties fbp;

fbp.color_depth= RGB_24BITS;
fbp.height= 100;
fbp.width= 100;

/*set img structure according to raw image format*/
ImageProperties imgp;
imgp.color_depth= RGB_24BITS;
imgp.height= 16;
imgp.width= 16;

/*create two frame buffers*/
FrameBuffer fb = NewFrameBuffer(&fbp);
FrameBuffer ffb = NewFrameBuffer(&fbp);

/*add a raw image to fb*/
/*consider that raw_img is already declared somewhere in code*/
AddRawImage(fb,&fbp,raw_img,&imgp,0,0);

/*translate fb (20,20) coordenates and store it in ffb*/
Translate(ffb,fb,&fbp,20,20);

/*HERE PASS THE ffb VARIABLE TO THE PLATFORM DEPENDENT DRAW METHOD OR DEVICE CONTEXT*/

/*release allocated memory*/
DeleteFrameBuffer(fb);
DeleteFrameBuffer(ffb);
```

## 5.2 Scaling

Zoom-in duplicates the pixels according to the scale parameter. Consider Figure 3 that shows a $4 \times 4$ pixels image.

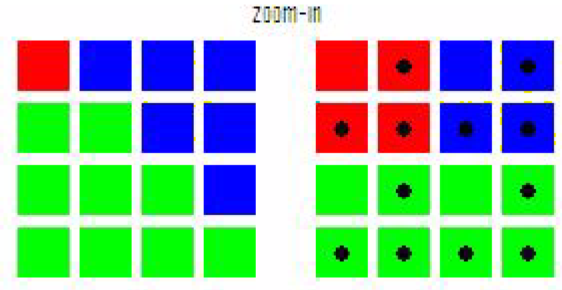


**Figure 3. Zoom-in Original and Scaled Images**

Using a scale parameter of 2, the scale method duplicates itself by two in both the axes. In Figure 3, the black circles point out the duplicated pixels. Whereas, zoom-out takes only some pixels from the original image and generates a new image based on the selected scale parameter.

For example, Figure 4 uses a scale parameter of 2 that takes only four samples of the original $4 \times 4$ pixels image. Black circles indicate the copied pixels.

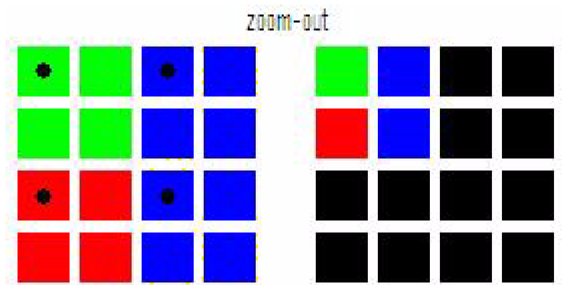


**Figure 4. Zoom-out Original and Scaled Images**

**NOTE**

This function is also dependent on the points that are centered around the upper left of the screen.

The `Scale` function takes the following format:

```
void Scale(FrameBuffer tfb,FrameBuffer fb, FrameBufferProperties* fbp, int32 scale);
```

**Example 4. `Scale` function**

```
/*set fbp structure according to frame buffer properties*/
FrameBufferProperties fbp;

fbp.color_depth= RGB_24BITS;
fbp.height= 100;
fbp.width= 100;
```

```
/*set img structure according to raw image format*/
ImageProperties imgp;
imgp.color_depth= RGB_24BITS;
imgp.height= 16;
imgp.width= 16;

/*create two frame buffers*/
FrameBuffer fb = NewFrameBuffer(&fbp);
FrameBuffer ffb = NewFrameBuffer(&fbp);

/*add a raw image to fb*/
/*consider that raw_img is already declared somewhere in code*/
AddRawImage(fb,&fbp,raw_img,&imgp,0,0);

/*zoom in fb by 2 and store it in ffb*/
Scale(ffb,fb,&fbp,2);

/*HERE PASS THE ffb VARIABLE TO THE PLATFORM DEPENDENT DRAW METHOD OR DEVICE CONTEXT*/

/*release allocated memory*/
DeleteFrameBuffer(fb);
DeleteFrameBuffer(ffb);
```

## 5.3 Rotation

All rigid body movements are either rotation or translation or combinations of the two. A rotation is simply a progressive radial orientation to a common point. This common point lies within the axis of that motion. The axis is 90º perpendicular to the plane of the motion. Mathematically, rotation is a rigid body movement that has a fixed point, unlike the translational motion.

**NOTE**

This definition applies to rotations in two and three-dimensions (motion in plane and space, respectively).

The `Rotate` function takes the following format:

```
void Rotate(FrameBuffer tfb, FrameBuffer fb, FrameBufferProperties* fbp, int32 angle);
```

**Example 5. `Rotate` function**

```
/*set fbp structure according to frame buffer properties*/
FrameBufferProperties fbp;

fbp.color_depth= RGB_24BITS;
fbp.height= 100;
fbp.width= 100;

/*set img structure according to raw image format*/
ImageProperties imgp;
imgp.color_depth= RGB_24BITS;
imgp.height= 16;
imgp.width= 16;

/*create two frame buffers*/
FrameBuffer fb = NewFrameBuffer(&fbp);
FrameBuffer ffb = NewFrameBuffer(&fbp);
```

```
/*add a raw image to fb*/
/*Let's consider that raw_img is already declared somewhere in code*/
AddRawImage(fb,&fbp,raw_img,&imgp,0,0);

/*rotate fb 30 degrees and store it in ffb*/
Rotate(ffb,fb,&fbp,30);

/*HERE PASS THE ffb VARIABLE TO THE PLATFORM DEPENDENT DRAW METHOD OR DEVICE CONTEXT*/

/*release allocated memory*/
DeleteFrameBuffer(fb);
DeleteFrameBuffer(ffb);
```

## 5.4 Shearing

Shearing is a process of sliding the pixels progressively on an image in a direction parallel to the x or y axis.

Figure 5 shows x axis and y axis shearing.

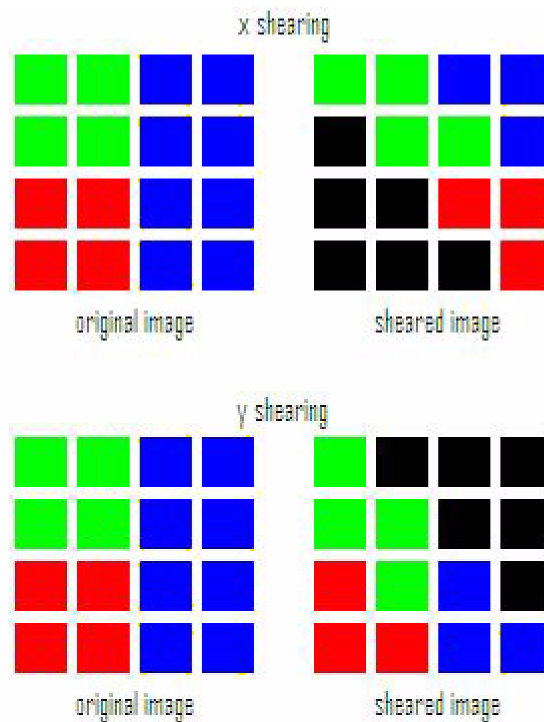


**Figure 5. x Axis and y Axis Shearing**

The `XShearing` and `YShearing` functions take the following format:

```
void XShearing(FrameBuffer tfb, FrameBuffer fb, FrameBufferProperties* fbp, int32 x);
void YShearing(FrameBuffer tfb, FrameBuffer fb, FrameBufferProperties* fbp, int32 y);
```

**Example 6. `XShearing` function**

```
/*set fbp structure according to frame buffer properties*/
FrameBufferProperties fbp;
```

```
fbp.color_depth= RGB_24BITS;
fbp.height= 100;
fbp.width= 100;

/*set img structure according to raw image format*/
ImageProperties imgp;
imgp.color_depth= RGB_24BITS;
imgp.height= 16;
imgp.width= 16;

/*create two frame buffers*/
FrameBuffer fb = NewFrameBuffer(&fbp);
FrameBuffer ffb = NewFrameBuffer(&fbp);

/*add a raw image to fb*/
/*consider that raw_img is already declared somewhere in code*/
AddRawImage(fb,&fbp,raw_img,&imgp,0,0);

/*shear fb and store it in ffb*/
XShearing(ffb,fb,&fbp,10);

/*HERE PASS THE ffb VARIABLE TO THE PLATFORM DEPENDENT DRAW METHOD OR DEVICE CONTEXT*/
/*release allocated memory*/
DeleteFrameBuffer(fb);
DeleteFrameBuffer(ffb);
```

# 6 Color Transformation

Digital color management requires translation of digital images between different representations or color spaces. For example, the pixels in an image may encode the colors that are visible when the image is displayed on a video monitor. At times, the details of the pixel transformation can be complex. Color is important in setting the mood of images and video sequences. Hence, color transformation is one of the most important features in photo editing or video postproduction tools. The `ColorTransform` function allows user to compensate the pixel color for red, green or blue channel.

The `ColorTransform` function takes the following format:

```
void ColorTransform(FrameBuffer fb, FrameBufferProperties* fbp, ColorProperties* cp);
```

**Example 7. `ColorTransform` function**

```
/*set fbp structure according to frame buffer properties*/
FrameBufferProperties fbp;

fbp.color_depth= RGB_24BITS;
fbp.height= 100;
fbp.width= 100;

/*set img structure according to raw image format*/
ImageProperties imgp;
imgp.color_depth= RGB_24BITS;
imgp.height= 16;
imgp.width= 16;

/*create a frame buffer*/
FrameBuffer fb = NewFrameBuffer(&fbp);
```

```
/*add a raw image to fb*/
/*consider that raw_img is already declared somewhere in code*/
AddRawImage(fb,&fbp,raw_img,&imgp,0,0);

ColorProperties cp;

cp.redMultiplier=1;
cp.redOffset=0;
cp.greenMultiplier=1;
cp.greenOffset=0;
cp.blueMultiplier=1;
cp.blueOffset=0;

/*apply color transformation*/
ColorTransform(fb,&fbp,&cp);

/*HERE PASS THE ffb VARIABLE TO THE PLATFORM DEPENDENT DRAW METHOD OR DEVICE CONTEXT*/

/*release allocated memory*/
DeleteFrameBuffer(fb);
```

## 6.1 Color to Grayscale

Grayscale images are the result of the intensity measurement of light at each pixel in a single band of the electromagnetic spectrum (infrared, visible light, ultraviolet, and so on). In such cases, they are monochromatic when only a given frequency is captured. Grayscale images can also be synthesized from a full color image.

The intensity of a pixel is expressed within a given minimum and a maximum inclusive range. This range is represented in an abstract way as a range from 0 (black) to 1 (white), with any fractional values in between. This notation is used in academic papers, and it must be noted that this does not define black or white in terms of colorimetry.

Figure 6 shows a grayscale palette.

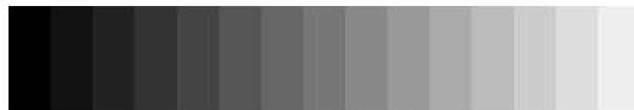

**Figure 6. Grayscale Palette**

Though the grayscale can be computed through rational numbers, the image pixels are stored in binary quantized form. Early gray scale monitors could show only up to 16 (4-bit) different shades. Today, grayscale images (as photographs) intended for visual display (both on screen and printed) are commonly stored with 8 bits per sampled pixel that allows 256 different intensities (that is, shades of gray) to be recorded on a non-linear scale. The precision provided by this format is barely sufficient to avoid visible banding artifacts and is convenient for programming, as a single pixel occupies only a single byte.

Whatever value is assigned to the pixel depth, the binary representations assumes the minimum value, 0 as black and maximum value (that is, 255 at 8 bpp, 65,535 at 16 bpp and so on) as white.

The `GrayScale` function takes the following format:

```
void GrayScale(FrameBuffer fb, FrameBufferProperties* fbp);
```

**Example 8. `GrayScale` function**

```
/*set fbp structure according to frame buffer properties*/
FrameBufferProperties fbp;

fbp.color_depth= RGB_24BITS;
fbp.height= 100;
fbp.width= 100;

/*set img structure according to raw image format*/
ImageProperties imgp;
imgp.color_depth= RGB_24BITS;
imgp.height= 16;
imgp.width= 16;

/*create a frame buffer*/
FrameBuffer fb = NewFrameBuffer(&fbp);
/*add a raw image to fb*/
/*consider that raw_img is already declared somewhere in code*/
AddRawImage(fb,&fbp,raw_img,&imgp,0,0);

/*Convert fb to grayscale*/
GrayScale(fb,&fbp);

/*HERE PASS THE ffb VARIABLE TO THE PLATFORM DEPENDENT DRAW METHOD OR DEVICE CONTEXT*/

/*release allocated memory*/
DeleteFrameBuffer(fb);
```

## 6.2 Color Filtering

Color filters are necessary because some camera sensors detect light intensity with little or no wavelength specificity. So, they are unable to separate the color information. The sensors are made of semiconductors, and they conform to solid-state physics. The color filter filters the light by wavelength range, such that the separate filtered intensities include information about the color of light. A demosaicing algorithm that is tailored for each type of color filter then converts the raw image data captured by the image sensor to a full color image (with intensities of all three primary colors represented in each pixel).

The `ColorFilter` function takes the following format:

```
void ColorFilter(FrameBuffer fb, FrameBufferProperties* fbp, ColorFilterProperties*
cfp);
```

**Example 9. `ColorFilter` function**

```
/*set fbp structure according to frame buffer properties*/
FrameBufferProperties fbp;

fbp.color_depth= RGB_24BITS;
fbp.height= 100;
fbp.width= 100;

/*set img structure according to raw image format*/
ImageProperties imgp;
imgp.color_depth= RGB_24BITS;
imgp.height= 16;
```

```
imgp.width= 16;

/*create a frame buffer*/
FrameBuffer fb = NewFrameBuffer(&fbp);

/*add a raw image to fb*/
/*consider that raw_img is already declared somewhere in code*/
AddRawImage(fb,&fbp,raw_img,&imgp,0,0);
/*set cfp structure according to color filtering parameters*/
ColorFilterProperties cfp;

cfp.red_max=100;
cfp.red_min=0;

cfp.green_max=155;
cfp.green_min=20;

cfp.blue_max=240;
cfp.blue_min=10;

/*apply color filtering*/
ColorFilter(fb,&fbp,&cfp);

/*HERE PASS THE ffb VARIABLE TO THE PLATFORM DEPENDENT DRAW METHOD OR DEVICE CONTEXT*/
/*release allocated memory*/
DeleteFrameBuffer(fb);
```

## 6.3 Color Inversion

Color inversion causes a change to all the colors in an image. In case of RGB color model, the inverse value is determined by subtracting the value of a color from the maximum color value. A numeric example for color inversion is given below:

A pixel has the values R (red) = 55, G (green) = 128, and B (blue) = 233 in a color resolution of 255 (8-bit color depth).

The inversion of this pixel is shown in Equation 2, Equation 3, and Equation 4:

$$R = 255 - 55 = 200 \quad \textbf{Eqn. 2}$$

$$G = 255 - 128 = 127 \quad \textbf{Eqn. 3}$$

$$B = 255 - 233 = 22 \quad \textbf{Eqn. 4}$$

The `Invert` function takes the following format:

```
void Invert(FrameBuffer fb, FrameBufferProperties* fbp);
```

**Example 10. `Invert` function**

```
/*set fbp structure according to frame buffer properties*/
FrameBufferProperties fbp;

fbp.color_depth= RGB_24BITS;
fbp.height= 100;
fbp.width= 100;
```

```
/*set img structure according to raw image format*/
ImageProperties imgp;
imgp.color_depth= RGB_24BITS;
imgp.height= 16;
imgp.width= 16;

/*create a frame buffer*/
FrameBuffer fb = NewFrameBuffer(&fbp);

/*add a raw image to fb*/
/*consider that raw_img is already declared somewhere in code*/
AddRawImage(fb,&fbp,raw_img,&imgp,0,0);

/*apply color inversion*/
Invert(fb,&fbp);

/*HERE PASS THE ffb VARIABLE TO THE PLATFORM DEPENDENT DRAW METHOD OR DEVICE CONTEXT*/

/*release allocated memory*/
DeleteFrameBuffer(fb);
```

## 6.4 Alpha Blending

Alpha compositing or alpha blending is the process of combining image with a background, to create an appearance of partial transparency. Alpha blending is useful to render image elements in separate passes and combine the resulting multiple two-dimensional images into a single final image. This process is called compositing. Compositing is used extensively when combining computer rendered image elements with live footage.

To combine the image elements correctly, it is necessary to keep an associated matte for each element. The matte contains the coverage information (that is, the shape of the geometry being drawn) to distinguish between the other parts of the image, where the geometry is actually drawn and other parts of the image that are empty.

As an example, the over operator can be accomplished by applying the formula shown in Equation 5 to each pixel value:

$$Value = (1 - \alpha)\ Value_0 + Value_1 \qquad \textbf{Eqn. 5}$$

The value of $\alpha$ in the color code ranges from 0.0 to 1.0, where 0.0 represents a fully transparent color, and 1.0 represents a fully opaque color.

The `AlphaBlending` function takes the following format:

```
void AlphaBlending(FrameBuffer background, FrameBuffer fb, FrameBufferProperties* fbp,
uchar8 alpha);
```

**Example 11. `AlphaBlending` function**

```
/*set fbp structure according to frame buffer properties*/
FrameBufferProperties fbp;

fbp.color_depth= RGB_24BITS;
fbp.height= 100;
fbp.width= 100;
```

```
/*set img structure according to raw image format*/
ImageProperties imgp;
imgp.color_depth= RGB_24BITS;
imgp.height= 16;
imgp.width= 16;

/*create two frame buffers*/
FrameBuffer fb = NewFrameBuffer(&fbp);
FrameBuffer ffb = NewFrameBuffer(&fbp);

/*add a raw image to fb*/
/*consider that raw_img is already declared somewhere in code*/
AddRawImage(fb,&fbp,raw_img,&imgp,0,0);

/*add a raw image to ffb*/
/*consider that raw_img2 is already declared somewhere in code*/
AddRawImage(fb,&fbp,raw_img2,&imgp,0,0);

/*apply Alpha Blending*/
AlphaBlending(ffb,fb,&fbp,alpha_70);

/*HERE PASS THE ffb VARIABLE TO THE PLATFORM DEPENDENT DRAW METHOD OR DEVICE CONTEXT*/

/*release allocated memory*/
DeleteFrameBuffer(fb);
DeleteFrameBuffer(ffb);
```

# 7 Drawing on Screen in Windows® CE

The following sections explain the procedure to draw on the screen in Windows CE platform, which is different from drawing in a Linux® platform. The image processing API delivers the processed frame buffer to implement the frame in Linux. The user should be familiar with drawing on a Linux screen. Though, Windows CE does not support full Win32 graphics API, the functions that the Windows CE supports allow the developers to write full featured graphical applications. There is no workaround for the features that the Windows CE does no support.

## 7.1 Painting Basics

Windows is divided into three main components:

- Kernel—handles the process and memory management
- User—handles windowing interface and controls
- Graphic device interface (GDI)—performs the low level drawing.

In Windows CE, the user and GDI are combined into the Graphic Windowing and Event subsystem (GWE).

## 7.2 Valid and Invalid Regions

When an area of a window is exposed to the user, that area or region is marked as invalid. When there are no messages in the application's message queue and the application's window contains an invalid region,

then the Windows sends a WM_PAINT message to the window. Any drawing performed in response to a WM_PAINT message is couched in calls to `BeginPaint` and `EndPaint`.

BeginPaint performs the following actions:

1. `BeginPaint` hides the text entry cursor, if it is not displayed
2. A WM_NCPAINT message is sent directly to the default window procedure, if required
3. Acquires a device context that is clipped to the invalid region
4. Sends a WM_ERASEBACKGROUND message, if required to redraw the background
5. Returns a handle to the device context

## 7.3 Device Contexts

Device Context (DC) is a tool that the Windows uses to manage access to the display and printer. Windows applications do not write directly onto the screen. Instead, they request a handle to a display DC for the appropriate window and then by using the handle, they draw to the context of the device. Windows then arbitrates and manages to get the pixels from the DC to the screen.

## 7.4 Windows CE Implementation

`WinMain` is the entry point. It must register a window class for the main window, create the window, and provide a message loop to process the messages for the window. In this method, the user can add the initialization code for the image processing API.

`WinMain` is prototyped as shown below:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,LPTSTR lpCmdLine,int
nCmdShow);
```

The messages sent or posted to the main window are sent to the `WndProc` procedure. `WndProc`, like other window procedures, is prototyped as shown below:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);
```

The function and their parameters are described as follows:

- `LRESULT` return type—is long and is entered in a way to provide a level of indirection between the source code and machine.
- `CALLBACK` type definition—specifies the function as an external entry point into the EXE. This is necessary as the Windows calls the procedure directly. The `CALLBACK` type definition varies depending on the targeted version of the Windows. It typically indicates that the parameters are pushed onto the stack in a right to left manner.
- `HWND hWnd`—is the window handle and is useful to define a specific instance of the window.
- `UINT message`—indicates the message being sent to the window. It is an unsigned integer containing the message value.
- `WPARAM wParam, LPARAM lParam`—are used to pass message specific data to the window procedure. In Windows CE, as in other Win32 operating systems, both the wParam and lParam parameters are 32 bits wide.

**Example 12. A part of the `WndProc` procedure implementation**

```
case WM_PAINT:

GetClientRect(hWnd,&rect);

hdc = BeginPaint(hWnd, &ps);

/*MY CODE STARTS HERE*/

/*it creates a memory device context (DC) compatible with the specified device*/
hdcMemory = CreateCompatibleDC( hdc );

/*it creates a bitmap with the specified width, height, and color format*/
hBmp = CreateBitmap(fbp.width,fbp.height,1,24,fb);

/*it selects an object into a specified device context. The new object replaces the
previous object of the same type. */
hOldSel= SelectObject( hdcMemory, hBmp );

/*The GetObject function retrieves information for the specified graphics object.*/
GetObject( hBmp, sizeof(BITMAP), &bmp );

/*This function transfers pixels from a specified source rectangle to a specified
destination rectangle*/
BitBlt(hdc,0,0,bmp.bmWidth,bmp.bmHeight,hdcMemory,0,0,SRCCOPY);

/*it selects an object into a specified device context*/
SelectObject(hdcMemory, hOldSel);

/*it draws text on the screen*/
DrawText(hdc,TEXT("TOUCH THE SCREEN TO START DEMO"),-1,&rect,DT_CENTER|DT_SINGLELINE);

/*it deletes device context handler*/
DeleteDC(hdc);

/*MY CODE ENDS HERE*/

EndPaint(hWnd, &ps);

break;
```

# 8 Conclusion

This application note helps to rapidly implement image processing to other software applications. The image processing API supports only 24-bits color depth. However, other color depths, such as 8-bit or 16-bit can be implemented easily. The API is endianess dependent. However, a function can be used to check for the endianness during the run-time. This way, the users can make their code more portable and flexible. In some functions, fixed point is implemented instead of floating point to avoid processor overhead. Since the API computes the image processing in a generic frame buffer, it can be used in Windows, Linux or any other OS theoretically.

# 9 Revision History

Table 1 provides a revision history for this application note.

**Table 1. Document Revision History**

| Rev. Number | Date | Substantive Change(s) |
|---|---|---|
| 0 | 05/2010 | Initial release. |

THIS PAGE INTENTIONALLY LEFT BLANK

*How to Reach Us:*

**Home Page:**
www.freescale.com

**Web Support:**
http://www.freescale.com/support

**USA/Europe or Locations Not Listed:**
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

**Europe, Middle East, and Africa:**
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

**Japan:**
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

**Asia/Pacific:**
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

**For Literature Requests Only:**
Freescale Semiconductor
Literature Distribution Center
1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com



Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale, the Freescale logo, CodeWarrior, ColdFire, PowerQUICC, StarCore, and Symphony are trademarks of Freescale Semiconductor, Inc. Reg. U.S. Pat. & Tm. Off. CoreNet, QorIQ, QUICC Engine, and VortiQa are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. ARM is the registered trademark of ARM Limited.
© 2010 Freescale Semiconductor, Inc.



Document Number: AN4042
Rev. 0
05/2010