Double Buffer Extension

By Jeanne Sparlin and Michael A. Cohen

Double Buffer Extension (DBE) is an Application Programming Interface (API) that allows developers to write software that provides smooth animation. This article provides hints for using DBE in applications.

The Double Buffer Extension is an X Consortium X Server protocol extension and Application Programming Interface (API) that allows application developers to write software that provides smooth animation. The Double Buffer Extension became an X Consortium standard in May 1995. This article discusses Double Buffer Extension Protocol Version 1.0.

Types of Buffers

A high-level understanding of graphics hardware is necessary to understand the importance of double buffering. In the simplest case, a graphics adapter contains a single color frame buffer that gets scanned to a monitor for viewing. Complex graphics adapters can have multiple sets of special purpose buffers, such as depth, stencil, alpha, and accumulation buffers, as well as multiple sets of color buffers. Special purpose buffers, which are not visible, are used in many rendering techniques. Some graphics adapters contain sets of color buffers that can be configured in several ways. For example, there might be 24 bitplanes of color buffer that can be used as 24 bits of a single color buffer or as two 8-bit color buffers—one that is visible and one that is hidden. (*Note:* When there is a single 24-bit frame buffer or two 8-bit frame buffers, 8 bits are left over.)

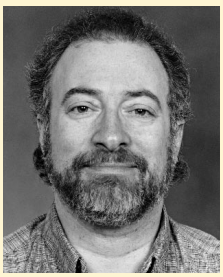
Multiple color buffers are used to produce smooth animation. An application draws to the back buffer while it is hidden. When the rendering is complete, the application swaps buffers. The frame buffer with the newly drawn picture becomes visible, and the formerly displayed buffer becomes invisible. The user instantly sees the new image without any artifacts of the actual rendering, such as tearing or the partial display of an image.

The technique of double buffering is analogous to a child's cartoon flip book that contains several pictures. In the flip book, the character's hand may move slightly in each frame to give the illusion that the character is waving. The same effect can be obtained by rendering each frame of the flip book in the back buffer, then swapping the back buffer to the front.

Buffer swapping was only available using the 3-D APIs in the initial releases of AIXwindows® and its associated 3-D APIs. The 3-D APIs in AIXwindows used a technique called *direct window access*. Rather than using the X Protocol to render, X Protocol reserves an area on the screen, then renders directly to the graphics adapter. Since the 3-D applications were a separate process from the X Server, the X Server was unaware that any buffer other than the one it was using was visible. This created a problem for applications that used the `GetImage` protocol to provide a screen capture facility. When the `GetImage` protocol request was received, the X Server grabbed the image from its buffer. So `GetImage` failed about 50% of the time because an alternate buffer was displayed.

In AIX 3.2.5, a fix for `GetImage` to the X Server combined with the MultiBuffer Extension (MBX) from the X Consortium provided a partial fix. The X Server now knew the currently displayed buffer and was able to retrieve `GetImage` data from the correct buffer. Since MBX was never an X Consortium standard, vendors implemented different prereleases of the extension. This resulted in confusion about the

Jeanne Sparlin



Michael A. Cohen

Library Call	Function
XdbeQueryExtension	Returns the major and minor version numbers of the DBE extension
XdbeGetVisualInfo	Returns information about which visuals support DBE and their relative performance
XdbeFreeVisualInfo	Frees storage allocated by XdbeGetVisualInfo
XdbeAllocateBackBuffer	Allocates a drawable ID that refers to the back buffer of a window
XdbeSwapBuffers	Swaps the buffers for all windows listed, applying the appropriate swap action for each window
XdbeGetBackBufferAttributes	Returns information about a back buffer

Figure 1. Primary library calls

semantics of some protocols and availability of some functionality.

MBX also uses an absolute method for displaying buffers. The application must explicitly draw in a specified buffer and keep track of what buffer is displayed. Absolute buffering did not meet the requirements of some 3-D APIs.

MBX was difficult to use and some supplied protocols, such as CreateStereoBuffers, were never implemented by any vendor. MBX was eventually discarded by the X Consortium and replaced by the Double Buffer Extension.

DBE uses relative buffering. When an application performs a swap buffer, the back buffer becomes the front buffer and the front buffer becomes the back buffer. The back buffer is always hidden and the front buffer is always displayed regardless of how many times a swap has occurred.

The DBE Interface

The Double Buffer Extension to the X Window System® provides a standard interface to applications for buffer creation, deletion, and swapping. The Double Buffer Extension provides the following functionality[1]:

- Allocates and deallocates double buffering for a window
- Draws to and reads from the front and back buffers associated with a window
- Swaps the front and back buffers associated with a window
- Specifies swap actions to be taken when a window is swapped
- Requests that the front and back buffers associated with multiple double-buffered windows be swapped simultaneously

X client applications access DBE through the Xext extension interface library. Figure 1 lists the primary library calls, describes their function, and explains their use.

Querying Extension Version

Extensions to the X protocol, like the X protocol itself, have major and minor version numbers. When protocol changes result in upwardly compatible software, the minor protocol number is increased. When protocol changes result in incompatible software, the major protocol number changes. Applications should query the major and minor protocol number when using X extensions to be sure that they can compatibly access the X Server portion of the extension.

An application should issue the XdbeQueryExtension before making any other DBE calls. This verifies that the application can compatibly access the X Server portion of the DBE extension and initializes the extensions.

If the application does not call XdbeQueryVersion, all subsequent DBE calls are undefined. This routine will return a non-zero value if the DBE library is compatible with the version returned by the X server. Zero is returned if the display does not support DBE or if there are other kinds of errors such as a communication failure with the server.

The sample program fragment in Figure 2 begins by making the XdbeQueryExtension call in order to start the use of double buffering: XdbeQueryExtension.

[1]Elliot, Ian (Hewlett-Packard) and Wiggins, David P. (X Consortium). *Double Buffer Extension Specification Protocol Version 1.0: X Consortium Standard.* 1995.

Creating Windows Available for Double Buffering

Once the application knows that the DBE extension is available and the version is compatible, the application must create a window that can be double buffered. The application first determines the appropriate Visual, an X Window System resource that gives hints about the adapter hardware.

The `XdbeGetVisualInfo` subroutine returns a list structure that describes the buffering capabilities for the X Server. It can be called for one screen or for all screens in a multihead configuration. Each screen that the application wishes to query can be specified by a drawable, such as a window or pixmap. Typically, the application uses the root window of each screen as the drawable passed to `XdbeGetVisualInfo`. The code fragment in the next segment queries one screen. It uses the default root window of the display as the drawable that it passes to `XdbeGetVisualInfo`.

The following are returned for each visual that provides double buffering capability:

- Screen (useful in multihead situations)
- Depth of the visual
- Performance level of the visual

The visual with the highest performance level is likely to have better double-buffer graphics performance than the visual with the lower performance level. The returned performance level is only useful when comparing visuals for a specific graphics adapter. An application can make no assumptions about double-buffer graphics performance from the size of the difference of the performance levels or by comparing performance levels of various graphics adapters.

In AIX®, `XdbeGetVisualInfo` reports two performance levels: a high and a low value. The high value indicates that hardware buffers are available for this visual and a low value indicates that software buffers are available for this visual.

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include </X11/extensions/Xdbe.h>

...additional includes

/* Global variables */
Display  *dpy;

...additional global declarations

...function definitions

main (int argc, char *argv[]){

...declarations
int majorVersion
int minorVersion;
char *ProgName;

...parse command line args
...open display

/* see if we can use DBE */
if ( !XdbeQueryExtension (dpy, &majorVersion, &minorVersion) ) {
  fprintf (stderr, "%s: XdbeQueryExtension() failed.\n", ProgName);
  exit (8);
}
```

Figure 2. Sample program fragment using DBE

```
...declarations
Drawable              screen_list [1];
int                   num_screens = 1;
XdbeVisualInfo        *visInfo;

screen_list [0] = DefaultRootWindow (dpy);

if ( (visInfo = XdbeGetVisualInfo (dpy, screen_list, &num_screens) ) == NULL) {
  fprintf(stderr, "XdbeGetVisualInfo returned NULL\n");
   ...handle error condition
}

...application decides which visual to use

XdbeFreeVisualInfo (visInfo);

...application creates the windows that will be used for double buffering
```

Figure 3. Sample program fragment

Usually an application uses the highest performance visual that will support the requirements of the application.

After the visual IDs for visuals supporting double buffers are determined, they can now be used with other subroutine calls (for example, `XGetVisualInfo` and `XMatchVisualInfo`) that help an application decide what visual type is best. In addition to the ability to double buffer, when applications are selecting visuals, they usually consider the depth and the color class of the visuals. Color class for the X Window System includes grayscale, pseudocolor, direct color, and true color. The selected visual is then used in the `XCreateWindow` subroutine call.

Once the visual is selected, the `XdbeFreeVisualInfo` call is used to free storage returned by `XdbeGetVisualInfo`. In Figure 3, the sample program fragment makes the call to `XdbeGetVisualInfo`, selects the best visual to use based on application and performance requirements, then frees the storage associated with the query.

Creating Double Buffers

To create a back (hidden) buffer for the window, the application calls the `XdbeAllocateBackBufferName` subroutine. When calling `XdbeAllocateBackBuffer`, the application uses the window that is created with the visual selected using the information from the `XdbeGetVisualInfo`. The window becomes the front or displayed buffer. The `XdbeAllocateBackBufferName` subroutine returns an X Window System resource called a back buffer (hidden buffer).

A back buffer is a new resource type defined by the Double Buffer Extension. In addition to window and pixmap, a back buffer becomes a drawable type. A back buffer can be used in any of the core X Window System rendering subroutines that accept a drawable parameter. Back buffers have the same depth and color class as well as the same clip list as their associated window. If the window is window shaped by using the X Non-Rectangular Window (Shape) Extension, the back buffer has the same shape as the associated window.

The application can begin drawing to this second or back buffer after a successful call to `XdbeAllocBackBufferName`. The sample program fragment is continued with the allocation of the back buffer and the building of data into it.

Swapping Buffers

During normal processing, an application would draw a scene into the back buffer and then call `XdbeSwapBuffers` to make the back buffer visible. An application might even associate some type of timing loop to the draw/buffer swap actions if it desires a smooth animation.

`XdbeSwapBuffers` will swap front and back buffers according to the swap action for each window listed in the array of windows passed to

```
...declarations
XdbeSwapInfo swapInfo;
Window win;
XdbeBackBuffer buf;

win = XCreateWindow (dpy,.........);

swapInfo.swap_action = XdbeBackground;
if ( buf = XdbeAllocateBackBufferName (dpy, win,swapInfo.swap_action) == None) ) {
  fprintf (stderr, "%s: Couldn't create buffers\n", ProgName);
  exit(1);
}   else   {
  /* Initialize swap window for later use */
  swapInfo.swap_window = win;
}

/* Call function to initialize the first image in the displayed window */
draw_image (win);

...ready to start animation
```

Figure 4. Allocating the back buffer and drawing initial image

`XdbeSwapBuffers`. Note that each window swapped in a single call to `XdbeSwapBuffers` can have a different `swap_action`. The `XdbeSwapInfo` structure contains two fields:

- Window for which to swap buffers
- Swap action to be applied during that swap

A swap action specifies what the X Server should do with the front buffer when the buffers are swapped. The four types of swap actions defined by DBE (illustrated in Figures 5, 6, 7, and 8) are described below.

Undefined. The contents of the new back buffer become undefined. This may be the most efficient action since it allows the implementation to discard the contents of the buffer if necessary. The application is not guaranteed anything about the state of the back buffer after the swap.

An application might use this swap action if it desires fast swaps and does not care what happens to the back buffer. For example, a specialized application might always copy an existing image from off-screen storage into the back buffer before it begins rendering. This could be a landscape or some other static background.

The application does not care what happens to the new back buffer when it is swapped. It is going to overwrite the entire buffer anyway.

Background. The unobscured region of the new back buffer will be tiled with the window background. The background action allows devices to use a fast, clear capability during a swap. This action might be used if the application is not rendering the entire buffer and wants the rest of the buffer filled with the window background. In one atomic protocol the buffer is swapped and the new back buffer is cleared and made ready for the next rendering. The application saves time because it does not have to send another protocol to clear the back buffer before it can begin rendering.

Untouched. The unobscured region of the new back buffer will be unmodified by the swap. This swap action might be used if an application is only going to modify a portion of the scene. It does not have to erase or rerender part of the scene. It just enhances what is already there. For example, an animation of a

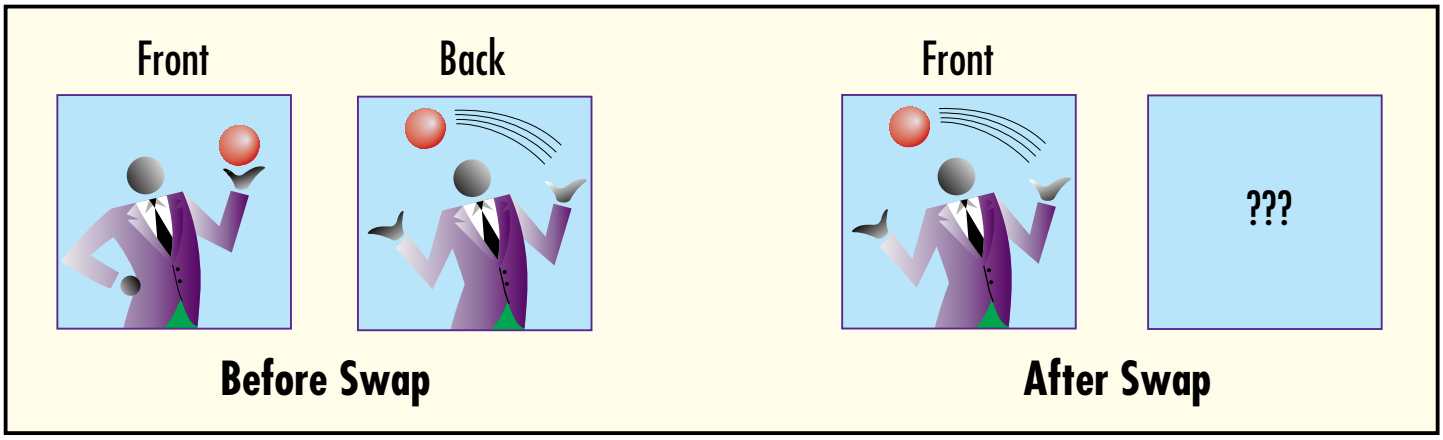


Figure 5. Undefined

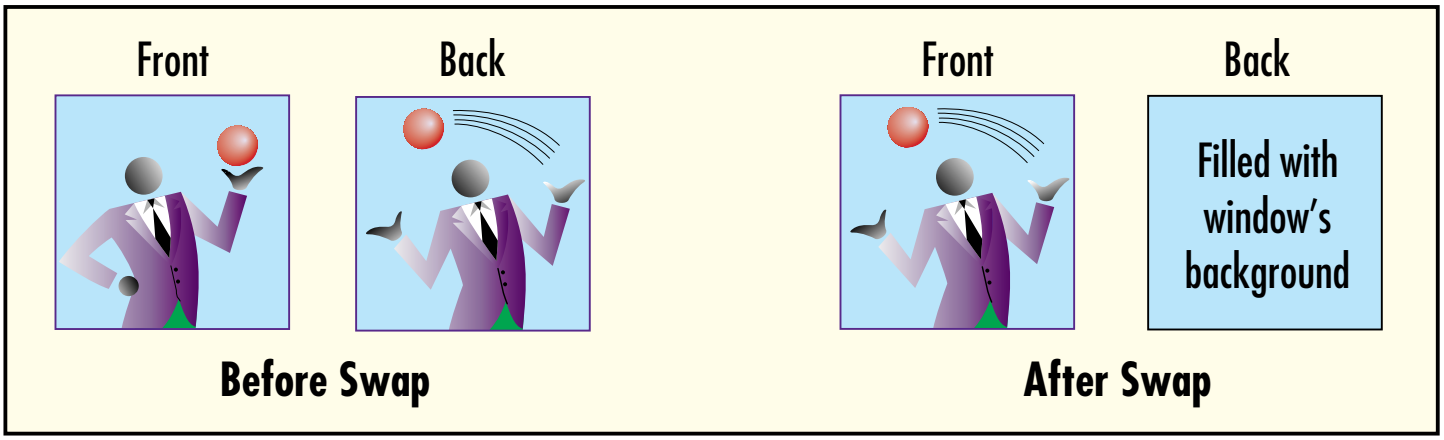


Figure 6. Background

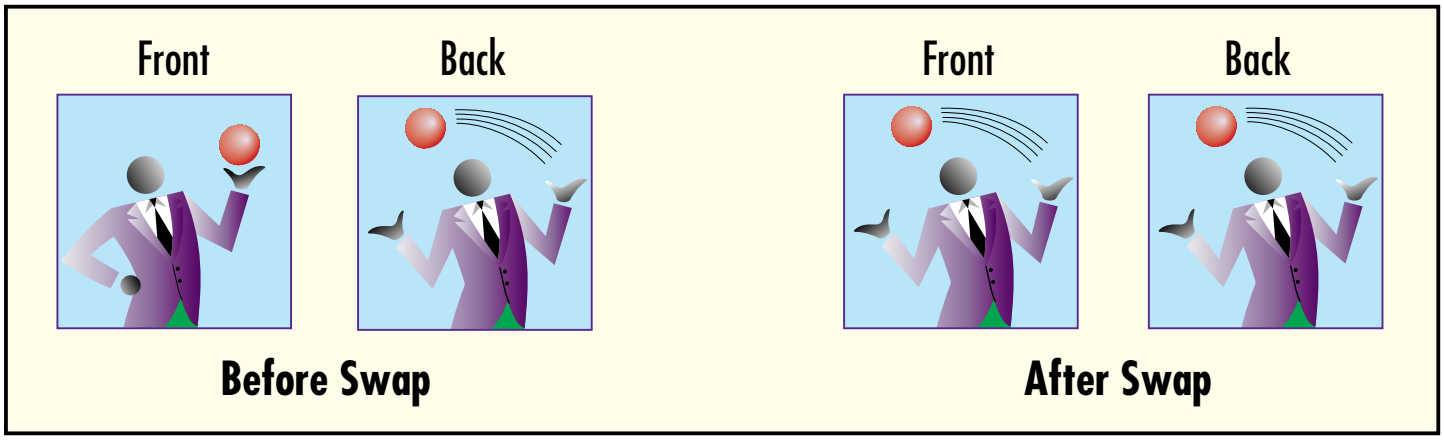


Figure 7. Untouched

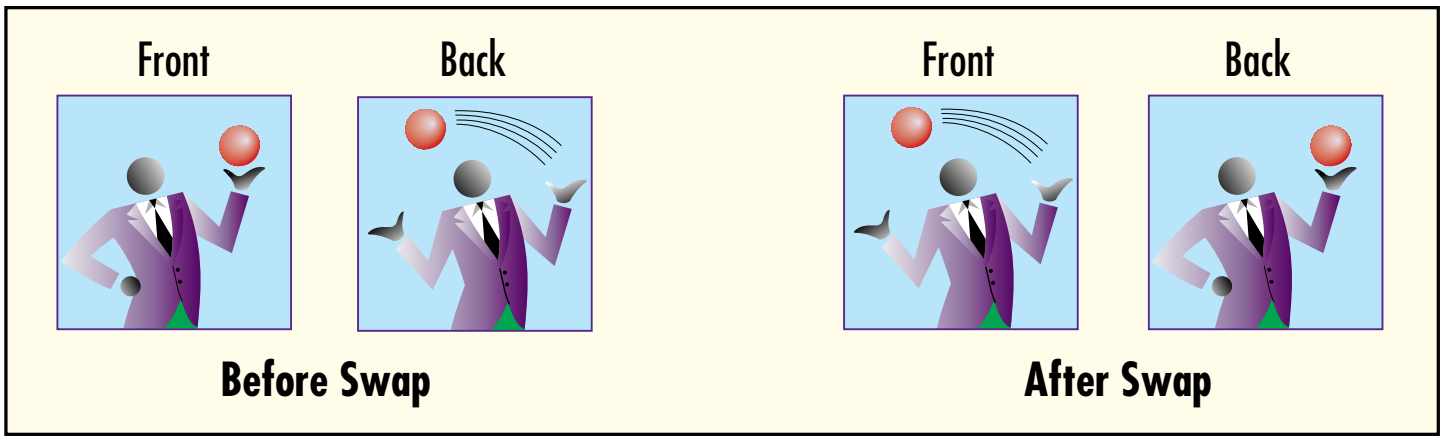


Figure 8. Copied

```
/* loop until some action ends the animation */
for (;;) {
  /* animate image by moving it slightly and drawing
     new image to the back buffer */
  recalculate_image ();
  draw_image (buf);
  XdbeSwapBuffers (dpy, &swapinfo, 1);
  if (we_are_done_animating())
    break;
}
```

Figure 9. Animating using XdbeSwapBuffer

house being built would add something new to each frame and not rerender the entire frame.

Copied. This swap action might be used if an application knows that it must maintain a high degree of data integrity. The newly exposed region will contain inconsistent data for any swap action besides Copied when a window is exposed. When the swap action is Undefined, there might be garbage; when the swap action is Background, the region will contain only a color or pixmap. Until the application can redraw the back buffer and swap, the newly exposed regions may look inconsistent. When the swap action is copied, the newly exposed regions remain consistent with the rest of the scene.

Figure 9 continues the sample program fragment with the use of `XdbeSwapBuffers`.

Conclusion

The Double Buffer Extension is a simple interface for software developers to incorporate smooth animation into their application programs. DBE provides an X Consortium standard API that was not present in previous versions of AIXwindows. DBE is available for both 2-D and 3-D developers. The Double Buffer Extension ships with AIX 4.1.4.

Jeanne Sparlin, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Ms. Sparlin joined IBM in 1983 after completing her computer science training at the University of Texas, Austin. She has previously worked in the Data Management, SNA, Dialog Design Aid, and the X Window System development departments on the RT PC®. She joined the Graphics Architecture department in October 1988 and currently does X Window System architecture. She also represents IBM on the Advisory Committee of the X Window System Consortium. She has a BS in Education from Northeast Missouri State University.

Michael A. Cohen, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Cohen joined IBM in 1988 and has participated in a variety of projects for the IBM graphics organization. He was a ddx programmer for the X Server for the GXT500 graphics adapter and, more recently, the lead developer for the Double Buffer Extension on several IBM RISC System/6000® platforms. He has a BS from Purdue University and a Master of Science in Computer Science from the New York Institute of Technology.

IBM Launches Virtual Pavilion

IBM has launched its "Small Planet Pavilion" Web site as part of the Internet's 1996 World Exposition. Expo '96 is a year-long event that will take place in cyberspace and in "real space" in schools, museums, and cultural centers around the world. Events will be held throughout the year on the Expo's "electronic fairground."

The IBM Small Planet Pavilion lets people experience the power of a networked world. The pavilion will be a participatory environment where individuals "learn by doing." The IBM Small Planet Pavilion can be found on the World Wide Web at http://www.ibm.park.org.