# Welcome to BackSpace!

This documentation is primarily aimed at programmers and advanced users. Most of the information required to use BackSpace should be in the information panel of the application.

BackSpace is mostly a screen-saver and screen-locking utility. It was designed to be extensible, so you can easily add your own screen saver modules. I created BackSpace primarily for 2 reasons:

- To prevent screen burn-in on an idle computer.
- To make our computers look flashy and cool even when they're doing nothing at all.

The whole project grew out of a simple application to install a flying starfield (a la Star Trek) as a backdrop for my normal work; hence the name BackSpace. I envision BackSpace as a community project; ideally many people will create and distribute inspired modules. The ones I supply offer only the slightest hint of what can be done with the NeXT computer.

 I would prefer that you distribute the source code to any screen saver modules you write, and that you make available source code to any application which is largely derivative of BackSpace.

That all having been said, I think BackSpace may often be a useful example, but don't think of it universally as a good example. It doesn't do anything illegal or undocumented per se, but it does use the application kit, window server, and low level event driver in ways they weren't necessarily designed to be used. It's more an example of brute-force, "you WILL do what I want" programming than elegant, "your code should look like this" programming. I try to point out inelegant sections of the code with comments, and I'll try to mention funky aspects of design in this document.

# Overview of a Screen Saver

At the lowest level, user actions are delivered to the window server by **ev**, the event driver. You don't have direct access to the **ev** driver, but you can poke at it via the **evs** driver. (3.0 note: This was true for 2.0, but the **evs** driver was kind of ugly. There's new high level functions that take care of this for you in 3.0, and the fellow that provided the new API was good enough to include give us functions that obviated most of the ugly hacks required in the previous BackSpace) The **ev** driver delivers events to the window server, which associates them with a window and a context and passes them along to the appropriate Application object, though these high level machinations are of little consequence to a screen saver. BackSpace queries the **evs** driver periodically to see if the user has done anything. When the computer has been idle for a period of time (settable via the Preferences application) BackSpace places a giant black window over all the other windows on the screen, effectively turning the screen completely black. Into this window, BackSpace installs the View object of your choice, and it periodically invokes the **oneStep** method, which you must write to take your View from the current animation frame to the next animation frame.

## Creating and Installing BackSpace

BackSpace is supplied in source code form in two directories, one for the core application including a few screen saver modules, and one for additional modules. To do a basic installation, type "make install" from a shell in the main BackSpace directory. This will produce a version of BackSpace in a folder named BackSpace.app in your Apps directory. (This version is not completely stripped, but it is stripped as much as possible to still allow run time linking.) Then change directories to the folder with the additional modules and type "make install" again to install the modules in this directory.

If you want to install BackSpace in a directory other than ~/Apps , you can specify the installation directory with the INSTALLDIR variable. For example, the following line will install BackSpace in /LocalApps:

```
make install INSTALLDIR=/LocalApps
```

The installed BackSpace directory (which will usually be ~/Apps/BackSpace.app) may contain several interesting things. First of all, it will probably contain a tiff or eps file called defaultImage.tiff or

defaultImage.eps. This image will be used by some of the screen saversr until the user selects another. That directory should also contain one or more .lproj directories. You can create a localized version of BackSpace by copying the English.lproj directory (renaming the copy to French.lproj, for example) By modifying the strings in the nib file and putting string equivalents in the Localizable.strings file, you can have a localized version of BackSpace without modifying the source code.

The Localizable.strings file contains all the strings compiled into BackSpace. To modify it, you would change a line in the form of "key" to "key" = "local Equivalent". For example,

```
        "All";
```
could become
```
        "All" = "Tout";
```
and
```
        "Original BackSpace String";
```
would become
```
        "Original BackSpace String" = "Translated BackSpace String";
```

## Writing a Screen Saver module

Writing a screen saver is simplified because BackSpace provides the user interface, the interface to the event driver, the black window, everything! All you need to provide is animation code.

Your screen saver module (an ordinary subclass of the View class) really only needs to implement two methods, the **oneStep** method and the **drawSelf::** method. When your view is on-screen, the BackSpace driver will invoke the **oneStep** method as fast as it can. Your implementation of this method should take the animation from the current frame to the next frame, typically by erasing the drawing at the current position and drawing something new at a new location. There are a number of additional methods that your View class may optionally implement to provide additional behavior, and you will probably want to override some of the standard View methods like **initFrame:** and **sizeTo:** to be aware of changes to the size of your View.

BackSpace supports both buffered and unbuffered windows, since both have advantages. True unbuffered windows (nonretained) are nice because you draw directly on the screen; they require almost no memory. Remember, a full screen buffer on a 24 bit color system requires 3 megabytes of memory! Full screen windows in BackSpace are non-retained by default, meaning they exist only in video memory. If the full size window is non-retained, the normal window will be a retained type. Retained windows are buffered as far as the Application kit is concerned, but are unbuffered as far as you are concerned; you draw directly on the screen wherever your window appears on screen, so your application sees no buffering. You may get better performance from an unbuffered window, since you don't draw things twice (once in a buffer, once on screen). On the other hand, it is considerably more difficult getting nice animations in an unbuffered window.

If your view implements the **useBufferedWindow** method and returns YES, it will be place in a buffered window. (Both the full size window and the normal window will be buffered.) Probably most complex animations will use a buffered window. However, if saving memory is of interest to you, I recommend looking into the code for the SpaceView and BoinkView classes, both of which are unbuffered. The SpaceView draws rapidly enough that erasing on screen and then drawing doesn't usually result in visible flicker. In fact, it looks much worse in a buffered window, since you must frequently flush the huge buffer to the screen. The BoinkView takes a different tack - it really needs some kind of buffer, though not a full screen buffer. I figure out the biggest buffer it will need and buffer that area myself in an NXImage. The animation gets buffered though the window is not.

Now some notes on non-retained windows. You already know I kind of like these since they take so little memory. Know too that I generally discourage their use since the application kit doesn't use them very well. In fact, I don't know of any other application that uses them heavily. When the window server needs to draw a regular window (Buffered or Retained) it just grabs the bits it buffered and redraws the window; it doesn't bother you at all. But it has no buffer to turn to in the non-retained case, so when one of these windows is exposed, it must ask the application to redisplay it. Unfortunately, this doesn't work very well; your application is asked to redraw the window even when it doesn't need it, which makes it look choppy sometimes. Also, sometimes the request to redraw the window is buffered up, and you don't receive the notification until the mouse moves. If it doesn't move, your window may just look like poop for

a while. The first problem can easily be dealt with in your code, see the **drawSelf::** description for more information. You'll just have to live with the second one; it doesn't come up too often and isn't crippling. I hate to special case things too much to dance around application kit flukes that will probably be solved in the future.

Here is a list of required methods, followed by a list of the optional methods:

**Required methods**

- **drawSelf**:(const NXRect *)*rects* :(int)*rectCount*;

You use this method to redraw your view. Your implementation of this method could be very simple if everything is moving, since even if you didn't draw everything it would appear as soon as it moved anyway. Some of my modules just use this to draw the expose area in black, which looks like this:

```
- drawSelf:(const NXRect *)rects :(int)rectCount
{
        if (!rects || !rectCount) return self;
        PSsetgray(0);
        NXRectFill(rects);
        return self;
}
```

In the unbuffered case, this makes the window sometimes look flickery, since it will be called at odd random times and black out the window when it doesn't need it. I take care of this case for unbuffered windows by filling them with black when I know they really need it, so you can ignore the the rectangle fill for these windows. Your animation will probably look better if you actually do use **drawSelf::** in the normal way to draw everything in the view (less the black background for unbuffered views). I seem to recall a synchronization problem between **drawSelf::** and **oneStep** that meant things drawn in **drawSelf::** sometimes didn't get erased in **oneStep**. If you get this behavior, you may just have to use a stripped out **drawSelf::** like the one shown above.

- **oneStep**

  You must implement this method to take your view from the current animation frame to the next. Look at some of the example modules to see how you might implement this method. This method will be called as rapidly as possible; the BackSpace driver has several optimizations which increase animation throughput. Note however that this method may be called too frequently; If the vertical blank rate is 68 hertz, it is useless to pump out more than 68 animation frames every second. If this is a problem for you, your code should address it. All my animations take considerably longer than this, so I'm not willing to pull my optimizations in order to put a cap on the maximum frame rate. You might check out the BackView class, which is an abstract superclass that provides a **timePassed:** method for animations that need to wait or go slowly. You might also check out the BoinkView class, which scales its animation based on the frame rate. The actual speed appears to be relatively constant, and the animation gets smoother on faster (or lightly loaded) machines.

  Before the **oneStep** method is invoked, focus is locked on your view, so you shouldn't **lockFocus** (it will only slow you down). After this method is called, the BackSpace driver will do an **NXPing()** and **flushWindow**, so you don't need to do those either (again, they will only slow you down).

**Optional Methods**

- **didLockFocus**

  This method will be called after focus is locked on your view. I use it to set up the graphic state for my Views without incurring that overhead in my **oneStep** method. More specifically, I use it to set the current font for my module. (I probably shouldn't need to do this, but I do.)

- (BOOL) **ignoreMouseMovement**

  If you respond to this method, you should return a BOOL to indicate whether BackSpace should exit screen saver mode in response to mouse movements. You could use this to create an interactive screen saver, such as a game that locks people out but still allows them to enjoy your NeXT. Maybe someday I'll use this to create a BoinkOut screensaver game...

- **inspector**:*sender*;

  If you implement this method, it should return a View containing any controls you want to present in the inspector section of BackSpace's settings panel. The best way to do this is to load a nib file containing the View the first time this method is called. If your nib has controls, you should load the nib such that the View is the file's owner, and in IB you should set the class of file's owner appropriately so that you can hook your controls to the appropriate outlets in file's owner (in other words, to the instance variables in your view). The inspector you return should be the same size as all the inspectors I provide. I don't remember that size, but you can start by simply copying one of my interface files and tailoring that inspector to your needs. *sender* will be the Thinker object if this message is sent from BackSpace (in other words, if you don't send it to yourself).

- **inspectorInstalled**

  This is a notification that your inspector has been installed. This will be useful if you present another window as an alternate inspector; for example, if you need more room than is provided in the settings panel.

- **inspectorWillBeRemoved**

  This is a notification that your inspector will be removed. This will be useful if you present another window as an alternate inspector. In this case, you should respond to this notification by ordering your alternate inspector out of the window list.

- (BOOL) **isBoringScreenSaver**

  If the use selects "All" in the settings panel, BackSpace will cycle through all the non-boring screen saver modules. If you implement this method and return YES, your module will not be selected unless the user explicitly selects it. This is useful for really boring screen savers like the one that turns the screen totally black, and for modules that are interesting as animations but not useful as screen savers (in other words, graphics that leave static images - which could burn in - on the screen. At least one screen saver must not be boring or BackSpace will hang (out of boredom, I suppose).

- **newWindow**

This informs the module that it or another member of its class has been potentially placed in a new window, and it should look at the size of the View. Generally, you should never implement this method, as your view can track changes in its size via the **initFrame:** and **sizeTo:** methods, and reset any instance variables accordingly. However, if you keep global information which is shared by several instances, you will need to reinitialize this information periodically. An example is the BezierView, which keeps its global state in the window server. (This is a bad practice, not to be encouraged! But it does make it easier to port cool PostScript code demos.) If you do implement this method, it should not output any drawing. (3.0 note: the latest version of BackSpace only instantiates 1 instance of each class of View, so you should never implement this method.)

- **setImage**: *newImage*

If you implement this method, it will be invoked to tell you when a new central image has been set. If you use this central image, you may wish to subclass the BackView class rather than the View class, since BackView has a few support routines to assist in the placement of an image.

- (BOOL) **useBufferedWindow**

If you implement this method and return YES, your view will always be place in a Buffered window. If you return NO, your View will be placed in an unbuffered window (which will be non-retained in the full screen case, otherwise retained.) By default, subclasses of View are place in unbuffered windows, and subclasses of BackView are placed in buffered windows. (This inconsistency is historical in nature.)

- (const char *) **windowTitle**

Use this method to return a window title for the normal window.

**Thinker Methods You Might Find Useful**

- (const char *) **appDirectory**

Returns the directory that BackSpace was launched from.

- **commonImageInspector**
  Returns BackSpace's common image inspector.

- (struct mach_header *) **headerForModule**:(const char *)*name*
  Returns the macho header returned by objc_loadModule() in the loading of the named module. I would imagine that you could use this header to load a nib of tiff segment, if you packaged up your .BackO in this manner. I haven't actually tested this feature to see if it works, however. For the FooView class, you would pass a *name* of "Foo"

- (const char *) **moduleDirectory**:(const char *)*name*
  Returns the directory that a module was loaded from. You can use this to find nibs in a .BackModule file package. For the FooView class, you would pass a *name* of "Foo"

## How to Add Your Screen Saver Module

The class-name for your module must end in *View*; for example *FireworksView*. You can either compile your module directly into BackSpace or compile it into a .BackO file and place it one of the right places, in which case it will be linked in at run time. To compile it in, add it and anything it needs to the Interface Builder project, and add the name of the class (without the *View* postfix) to the array at the top of the ThinkMore.m file.

If you don't want to compile your module in, you will need to compile it such that the name of the .BackO file is the same as the name of the screen saver View class it contains. You can then place your module in the folder that BackSpace was launched from (which is an app-wrapper, a folder that looks like an executable file from the WorkSpace Manager), in **~/Library/BackSpaceViews**, or in **/LocalLibrary/BackSpaceViews**.

There are some caveats to run-time linking. The load order for subclasses is critical; superclasses must be present when a class is loaded. It's difficult to ascertain the load order, so to be safe, your modules

should only be subclasses of View or of a class that's compiled into BackSpace. It's more difficult to debug classes linked at runtime, though if you don't need to use the debugger, it seems much quicker to leave modules external; compile and link time seem better. You can strip external modules somewhat, but they need global symbols. This means you can use the -x option to ld (which makes modules quite a bit smaller) but you can't use the -x option to strip (on the modules, that is.) Finally, you can link a bunch of .o files together to create a single module; I do this with .m and .psw files to create a complete module in a single .o file. However, the appkit classes will not search the mach headers of run time loaded modules for additional images or things. I suggest putting images into the app-wrapper rather than linking them into the .o file.

3.0 note: BackSpace now also accepts .BackModule files. A .BackModule is simply a file package, a folder containing a .BackO module and perhaps additional files such as nibs and TIFF images. The .BackModule folder must have the same name as the .BackO found within, so you could put a FooView.BackO module into a FooView.BackModule and it should work fine. BackSpace keeps a record of the load directory and macho header for each module, so a module can locate its resources based on its name. To do this, you must send an inquiry to the Thinker object, which is available via the **BSThinker()** function and is also the sender of **inspector:** messages. In the case of the FooView module, the code would look like this:

```
[(BSThinker()) moduleDirectory:"Foo"];
[(BSThinker()) headerForModule:"Foo"];
```

See the Thinker.h file for the details on this interface. (Incidentally, modules shouldn't assume that the Thinker object is NXApp's delegate; though this is true for the current version of BackSpace, it can't remain true if the BackSpace engine is loaded into other applications such as Preferences.)

## Changes from BackSpace 1.0

There are lots of small improvements to BackSpace since the original version. Here are the more significant ones:

- It now declares 2 file types, .BackO and .BackModule. The .BackO is exactly like the old .o format (you can rename all the old modules) but now I can recognize them by name and load them if you double click. The .BackModule is a file package containing a .BackO and any other things you need, like tiffs and nibs. The shell script **convertToBackOs** is provided to quickly rename all the old .o files for you, for easy conversion.
- BackSpace keeps the load directory and macho header for each module, and can return the path for each one, so they can find their associated files.
- BackSpace now allows each module to put an inspector in the settings panel, so you can easily configure each module.
- You now select modules using a browser rather than the old pop-up list.
- BackSpace now only instantiates one view of each class.
- A view can set a flag allowing it to ignore mouse movements, opening the door to game modules.
- BackSpace used to load the class for every module at launch time. Now it does both lazy class loading and lazy instantiation. In other words, loading and instantiation are deferred until you actually need a specific View.
- The evs driver finally has decent API, so I don't have to kludge around to force the screen dim, or to guess when it will go dim.

## Known bugs

(Mostly fixed in 3.0....)

In the Boink module, the grid lines look kind of spotty on a monochrome monitor. This is because I draw color lines, and they are dithered to the dither phase of the moving redraw area rather than the dither phase of the window. At some point I'll fix this by checking if the view **shouldDrawColor**, or by dynamically altering the dither phase.

## BackSpace user hints

BackSpace stores its screen locking password in the defaults database. If you ever forget this password,

you can nuke the old one by typing this into a shell:

```
dremove BackSpace encryptedPassword
```

The BackSpace password panel will also accept the user's password and the root password. I don't recommend using these passwords, however, because BackSpace could be easily modified to monitor passwords, and this would be **Very Bad**.

BackSpace will sort of fight with any other application that sets the screen brightness values or dim time (like Preferences). You should only fiddle with Preference's screen features when BackSpace isn't running. Nothing supremely wretched will happen, but BackSpace might not work very well as a screen saver... (By the way, I didn't add dim time adjustment to BackSpace because that's just one more way the two could fight. So set that dim time in Preferences!) If BackSpace autolaunches before Preferences, it might get confused. Actually, I haven't had a problem with it, though.

If you ever have to kill BackSpace to nuke the screen locker (by remotely logging in as root for example), the cursor will have disappeared. If you can get to a terminal program and run **pft**, you can get the cursor back by executing the **showcursor** PostScript operator.

## How to add commonly requested features

I make the screen bright when the screen saver kicks in so you can see the animations. If you wish the screen still went full dim, or if you wish to dim to some value other than normal brightness, modify the screenSaverMode method in ioctls.m. This method is invoked when the screensaver mode is enabled; if it does nothing then the screen will dim as normal.

Some people don't like the grid behind the ball in the Boink module. (By the way, they move slowly so they won't burn in.) Still, you can get rid of them by making sure that the **nvert** and **nhoriz** variables are 0; check out the **newViewSize** method in the BoinkSpaceView class.

## More Philosophy

The animated windows in BackSpace are all one-shot windows, which means that that the window server windows associated with those Window objects are thrown away any time they're not visible. In other words, if you can't see BackSpace, it's not taking up a lot of memory, even if you use a screen saver module that requires a big buffered window. One shot windows can be a pain in the rear; since the actual window doesn't always exist, you can't always execute PostScript code that depends upon its existence. (About the only time that's an issue is if you have to do something weird like alter its tier, which BackSpace does all the time...)

I always edit my files with my tab size set to 4 characters. The spacing will probably look funny if you use a different size.

That's all I can think of for now. Enjoy this program and go crazy with those modules!

-sam

Release History

| | |
|---|---|
| 1.01 | Beta versions, contain miscellaneous small bugs |
| 1.02 | First release? |
| 1.30 | Unreleased, first 3.0 API version? |
| 3.00 | Reworked for 3.0 IB, PB, event driver. Small bug fixes, ready for 3.0 release |