

Behind Bars

version 1

Michael S. Engber
Apple Computer - Newton ToolBox Group
Copyright © 1997 - Michael S. Engber

This article was (will be) published in the Newton Technology Journal. For information about becoming a Newton Developer email devsupport@applelink.apple.com or call (408)974-4897.

Introduction and Long Disclaimer

This article discusses various issues related to the screen orientation and the Button Bar in the most recent line of Newton devices. For example, on the MessagePad 2000 the Button Bar is drawn on the LCD screen. Therefore, it's possible to obscure it, reconfigure it, change its appearance, or entirely replace it.

The article describes how to do things that will put the Newton in a non-standard configuration which users may find confusing. Normal, well-written, applications should have no need to do this. This information is intended for applications with special needs and should be thought of more as techniques for customizing a particular Newton device rather than as general purpose APIs.

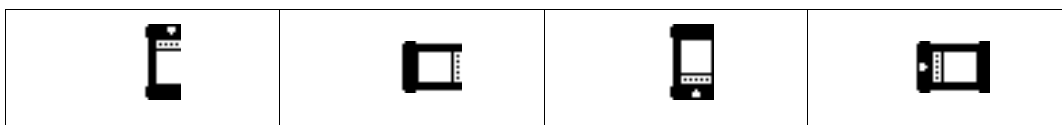
Do not give in to the temptation to add frivolous features to your application. Every application does not need a preference checkbox to put its icon on the Button Bar.

Furthermore, most of the information presented here is not officially supported and is not guaranteed to work on future Newton Devices. Applications that use it run the risk of incompatibility with future systems. Of course, efforts will be made to maintain compatibility, but changes in design may make compatibility impossible. For example, the Button Bar could be radically redesigned making many of the options below no longer applicable.

Despite this caveat, I'm sure there are still developers who will be anxious to use this information. By following the suggestions in this article you will maximize your application's chances for compatibility with future Newton devices. Each section ends with a compatibility note to explicitly summarize these considerations.

Changing the Screen Orientation

There has always been an API for getting the current screen orientation, `GetOrientation`. Now there is an API for changing the screen orientation, `SetScreenOrientation`. It takes one argument, an integer specifying the desired orientation (See Table 1).







| | | | |
|---|---|--|---|
|  |  |  |  |
| kPortrait 0 | kLandscape 1 | kPortraitFlip 2 | kLandscapeFlip 3 |

Table 1 - Screen Orientations for the MessagePad 2000 and eMate300

The return value of `SetScreenOrientation` is a `nil` or non-`nil` value indicating failure or success. For example, if the backdrop application is incompatible with the new orientation then the rotate will fail. Also note that `SetScreenOrientation` may present the user with a dialog giving the option of canceling the rotation because certain applications won't work in the new orientation. If the user cancels the operation `SetScreenOrientation` will return `nil`.

Compatibility Notes

- `SetScreenOrientation` was added in 2.1. Check for its existence using `GlobalFnExists` if your code needs to run on earlier ROMs.
- Remember to use `LegalOrientations` to get an array of the available orientations.
- Check the return value (or call `GetAppParams`) instead of assuming the unit is in the new orientation.
- Do **not** use the seductively named function, `SetOrientation`. It is unsupported and does not do what its name implies.

Moving the Button Bar

There are hooks in the system to allow changing the position of the standard Button Bar and the position of the controls, scroll arrows and overview button, within it. They are all controlled by `userConfig` values which are arrays of four elements, one element for each orientation. For example, `array[kPortrait]`, specifies the value for the portrait orientation.

To restore any of these settings to their default value set their entire `userConfig` value to `nil`. Individual orientations can be set use their default value by specifying `nil` for the corresponding array element.

`buttonBarPositions`

`buttonBarPositions` is an array of four elements which specify the location of the Button Bar. The allowed values for the elements are; the symbols: `top`, `left`, `bottom`, & `right`; and `nil`.

`buttonBarControlsPositions`

`buttonBarControlsPositions` is an array of four elements which specify the location of the scrolling and overview controls within the Button Bar. The allowed values for the elements are; the symbols: `top` & `bottom`, for when the Button Bar is on the right or left; the symbols `left` & `right`, for when the Button Bar is on the top or bottom; and `nil`.

`bellyButtonPositions`

`bellyButtonPositions` is an array of four elements which specify the location of the overview button relative to the scroll arrows. The allowed values for the elements are; the symbols: `outside`, `inside`, `left`, & `right`; and `nil`.

Compatibility Notes

- Make sure there is a soft Button Bar (if `GetRoot().Buttons.soft` is `True`) before setting these values.
- Make sure your positioning of the controls is consistent with the position of the Button Bar.
- Positioning the Button Bar so that the appArea becomes less than 320 high means that views without a `ReOrientToScreen` method will be unable to open – like the behavior of MessagePad 120 and 130 units in landscape orientation.

Covering the Button Bar

Applications that wish to cover the entire screen need to ensure they work in all screen orientations and all Button Bar positions – especially on MessagePad 2000 units in landscape orientation with the Button Bar on the left – the key point being the appArea having non-zero top-left corner.

Remember, children of the root view open relative to the appArea. This means using a *<left, top>* coordinate of *<0, 0>* will not cover the Button Bar unless it's on the right or bottom edge of the screen. In this situation the view's bounds need to be offset by the appArea's *<left, top>* global coordinates. `GetAppParams` now returns this information in the `appAreaGlobalLeft` and `appAreaGlobalTop` slots.

Depending on the reason for covering the entire screen, there are different approaches to use.

If the goal is simply to maximize the visible area of the base view then the Button Bar should be obscured only if it's located on the LCD screen as it is on the MessagePad 2000. On a MessagePad 130 the root view encompasses a larger area than the LCD screen – the tablet. Obviously, drawing is limited to the screen so applications don't increase their visible area by covering the Button Bar on a MessagePad 130.

Below is some sample code you can add to a base view's `viewSetupFormScript` to maximize visible area:

```
local params := GetAppParams();
if GetRoot().Buttons.soft then
    self.viewBounds :=
        OffsetRect( UnionRect(params.appAreaBounds,
                                params.buttonBarBounds),
                    -params.appAreaGlobalLeft,
                    -params.appAreaGlobalTop)
else
    self.viewBounds := params.appAreaBounds;
```

If the goal is to prevent users from accessing the buttons then the Button Bar should be obscured regardless of whether or not it is on the LCD screen.

On units like the MessagePad 130, remember to take into account the fact that part of the base view will be off-screen. For example, it's important to ensure that the close box is visible. A simple way to accomplish this is by having a child view whose bounds are the appArea and locating the rest of the application within that child.

Below is some sample code you can add to the base view's `viewSetupFormScript` to cover the entire tablet:

```

local params := GetAppParams();
self.viewBounds := GetRoot():LocalBox();
if params.appAreaGlobalLeft then
    self.viewBounds :=
        OffsetRect( self.viewBounds,
                    -params.appAreaGlobalLeft,
                    -params.appAreaGlobalTop)

```

Compatibility Notes

- On some units (e.g. the eMate 300) the buttons are not located in a view at all. Therefore, covering the entire tablet does not prevent the user from accessing the buttons (e.g. opening up the Extras Drawer).
- `appAreaGlobalLeft` and `appAreaGlobalTop` are new slots in `GetAppParams`. Check them before using them if your code has to run on older devices.

Closing the Button Bar

Applications that wish to replace the Button Bar need to use `KillStdButtonBar` to close it. Closing the Button Bar view directly will leave a hole – the `appArea` will not be readjusted to include the unused Button Bar area.

`KillStdButtonBar` is intended to accommodate Button Bar replacements. If your application simply wants to cover the entire screen `KillStdButtonBar` is **not** the correct way to accomplish this. Instead, open up a full screen view as described above.

After closing the Button Bar, it's assumed that your application will provide the user with a replacement (e.g. a floater) that will provide a way to do the following:

- scroll up and down
- overview
- open the Extras Drawer (from which the user can access former Button Bar icons)

Replacing the Button Bar does **not** mean you should replace the contents of the `buttons` root view slot with a reference to your view. It is important that you do not do this. The system relies on the `buttons` slot containing the Button Bar – whether it's open or closed. In fact, there is no need at all for the root view to have a slot referencing your view – you can create an open your view using `BuildContext`.

Your replacement view is private, the system will not send it message. It does not need to provide any of the Button Bar APIs described in this article. In order to be notified of changes (e.g. card yanking, package loading) you should register for changes on the "Packages" soup. This is an undocumented soup and you should definitely not rely on the format of its entries. When you are notified of a change, of any type, simply rebuild your list of icons.

To close the Button Bar you can use code like the following:

```
KillStdButtonBar(Array(4, '{buttonBarPosition: none}));
```

To restore the Button Bar use the following code:

```
KillStdButtonBar(nil);
```

It's also possible to reserve an edge of the screen for your replacement Button Bar so it can sit outside the appArea – like the standard Button Bar does. This would be useful if, for example, you create a replacement Button Bar that's thinner than the standard Button Bar.

The following code reserves the bottom twenty pixels in all four screen orientations:

```
KillStdButtonBar(Array(4, '{buttonBarPosition: bottom, buttonBarThickness: 20}'));
```

You can vary the position and or thickness of the reserved area in each orientation by varying the position and thickness of the corresponding element of the array passed to `KillStdButtonBar`.

`KillStdButtonBar` makes no attempt to arbitrate conflicts between applications. If two applications try to use `KillStdButtonBar` there's no way for one application to quit and restore the previous state of the Button Bar. All you can do is put back the standard Button Bar.

It is assumed that conflicts will be rare. It seems unlikely (undesirable) for users to have two different Button Bar replacements installed at once. Nevertheless, program defensively. Check if the Button Bar is open (call `kViewIsOpenFunc with(GetRoot().buttons)`). If it's not open inform the user that there's a conflict and don't call `KillStdButtonBar`.

Compatibility Notes

- Only use `KillStdButtonBar` if you're replacing the Button Bar, not if you want to cover it.
- `KillStdButtonBar` is a new API. Check for its existence using `GlobalFnExists` if your code needs to run on earlier ROMs.
- Configuring the Button Bar area so that the appArea becomes less than 320 high means that views without a `ReOrientToScreen` method will be unable to open – ^ la the behavior of MessagePad 120 and 130 units in landscape orientation.
- Stick to the APIs (`GetPartCursor`, `GetPartEntryData`, `GetPartEntries`) rather than accessing the entries in the "Packages" soup directly.

Configuring the Button Bar

We recommend letting users control what's in the Button Bar. It's a simple matter for users to drag icons in and out of the Button Bar themselves. Changing the Button Bar behind user's backs can confuse them. For example, an application that automatically installs itself in the Button Bar will have to push some other icon off causing the user to wonder where that icon went.

The mechanism by which icons are marked as being located in the Button Bar is simply filing – specifically, being filed in the `_ButtonBar` folder. This can be accomplished by changing an icon's `labels` slot using the Extras Drawer method, `SetExtrasInfo`. However, using `SetExtrasInfo` to move an icon to the Button Bar provides no control over its placement in the Button Bar.

There are two Button Bar methods that give you more control, `GetPartEntries` and `ReConfigure`. `GetPartEntries` takes no arguments and returns a frame with two slots, `fixed` and `mobile`. These slots contain arrays of part entries – ^ la the Extras Drawer method, `GetPartCursor`.

The fixed entries are “fixed” because they cannot be moved by dragging. By default, only the Extras Drawer's icon is fixed. It's important that the Extras Drawer icon be fixed. Users should not be able to drag the Extras Drawer icon into the Extras Drawer. The mobile entries are “mobile” because they can be dragged in and out of the Extras Drawer by the user.

As was previously stated, users should be in control of the contents of the Button Bar. We do not recommend making your application's icon fixed. Fixed icons are intended for use by licensees and VARs creating Newtons that are only used for specific purposes.

The ordering of the entries in the fixed and mobile arrays corresponds to the order of the icons in the Button Bar. The icons for the fixed entries are first, followed by the icons for the mobile entries.

As was previously mentioned, the elements of the fixed and mobile arrays are part entries. You should not examine them directly – just as you should not directly examine the entries returned by `GetPartCursor`. There is an Extras Drawer method, `GetPartEntryData`, that returns a frame of information about the entry (icon, text, `appSymbol`, `É`).

The Button Bar's `ReConfigure` method takes one argument, a frame of fixed and mobile entries, and reconfigures the Button Bar. The order of the entries in the arrays controls the order of the icons in the Button Bar. For convenience, `ReConfigure`, also accepts `appSymbols` instead of part entries. This allows an icon to be added or removed from the Button Bar without having to lookup its actual part entry.

A related Button Bar method that you may want to use in conjunction with `ReConfigure` is `IconCapacity`. `IconCapacity` takes no argument and returns the number of icons (fixed plus mobile) the Button Bar can currently hold. This number varies depending on the orientation and location of the Button Bar. It will be zero if the Button Bar is closed.

Compatibility Notes

- Make sure there is a soft Button Bar (`if GetRoot().Buttons.soft É`) before using these methods.
- Make sure the Extras Drawer Icon is fixed – to prevent it from being dragged into the Extras Drawer.
- Remember that whether or not Extras Drawer is the backdrop application affects whether or not it shows up in the Button Bar.
- Use `IconCapacity` to prevent overfilling (and those annoying notifications)
- Stick to the APIs (`GetPartCursor`, `GetPartEntryData`, `GetPartEntries`) rather than accessing the entries in the "Packages" soup directly.

Changing the Spacing and Font of Icons

There are hooks in the system to allow changing the font and spacing of the icons in the Extras Drawer and the Button Bar. They are all controlled by `userConfig` values.

```
buttonBarIconSpacingH  
buttonBarIconSpacingV
```

These two `userConfig` values control the spacing of the icons in the Button Bar. They are integers specifying the spacing in pixels. They both default to 40 in the MessagePad 2000. The vertical spacing (`buttonBarIconSpacingV`) is only used when the Button Bar is laid out vertically – along the right or left edge of the screen. The horizontal spacing (`buttonBarIconSpacingH`) is only used when the Button Bar is laid out horizontally – across the top or bottom edge of the screen. To restore either of these settings to their default value set their `userConfig` value to `nil`.

`extrasIconSpacingH`
`extrasIconSpacingV`

These two `userConfig` values control the vertical and horizontal spacing of icons in the Extras Drawer. They are integers specifying the spacing in pixels. They default to 64 horizontally and 52 vertically in the MessagePad 2000. They have no effect when the Extras Drawer is in overview mode. To restore either of these settings to their default value set their `userConfig` value to `nil`.

`extraFont`

This `userConfig` value controls the font used for the icon labels in both the Extras Drawer and the Button Bar. You should stick to the integer font specifications (e.g. (`userFont9 + tsPlain`) or (`simpleFont9 + tsBold`)). Using the integer representation in this instance accomplishes two things; it reduces NS Heap usage (non-default `userConfig` values occupy NS heap space) and it restricts you to the set of built-in fonts. Using a font that's not in ROM would be an extremely bad idea because the font could be removed. This information is stored in a soup. A user may be forced to do a cold-boot in order to remove a bogus font specification.

Compatibility Notes

- Stick to using built-in fonts for `extraFont` and specifying them in integer form.
- Remember to accommodate icons with two line titles when changing the Extras Drawer spacing.
- Make sure there is a soft Button Bar (`if GetRoot().Buttons.soft É`) before setting `buttonBarIconSpacingH` and `extrasIconSpacingV`.
- Setting `extraFont` and `extrasIconSpacingX` has no effect on earlier ROMs.

References and Suggested Reading

Dublin, Louis I., "Water Fluoridation: Facts, not Myths." , Public Affairs Pamphlet Number 251B, New York, The Public Affairs Committee. 2nd edition, 1967.

“In many American cities, a technical debate – *whether to raise the fluoride content of public drinking water as a dental health measure* – is attracting nearly as much attention as juvenile delinquency, education, automobile accidents, or the hydrogen bomb.”

Sharp, Maurice, Extra Extra: Extras Drawer Features in Newton 2.0. Newton Technology Journal, February 1996, pp. 1,17-89

This article discusses using part entries in the extras drawer. These are the same entities that are used in the Button Bar.