

# Miscellaneous

---

This chapter lists miscellaneous additions to the Newton OS introduced with Newton 2.1 OS that are not lengthy enough to warrant their own chapters. The following topics are covered in this chapter:

- The button bar interface
- The clipboard interface
- The dial-in networks interface
- Two protos for providing a password slip: `protoPasswordSlip` and `protoBlindEntryLine`
- Setting a transport preference for automatic put away of received items
- The format of Newton Works word processor soup entries
- A number of new user configuration variables: `LCDCContrast`, `alarmVolumeDb`, `soundVolumeDb`, `buttonBarPosition`, `buttonBarControlsPositions`, `bellyButtonPositions`, `buttonBarIconSpacingH`, `buttonBarIconSpacingV`, `extrasIconSpacingH`, `extrasIconSpacingV`, and `extraFont`.
- A new serial communication tool option: `kCMOPCMCIAModemSound`.

## Miscellaneous

- A number of new functions and methods:
  - **view:DragAndDropLtd** allows you to specify a limit bounds for drag and drop, and **view:ViewAddDragInfoScript** allows you to handle global command keys if your view supports cut and copy
  - **RegStationaryChange** (and **UnRegStationeryChange**) notifies you when a piece of stationery changes
  - **MakeFontMenu** creates a menu of available fonts suitable for a picker
  - **RecognizeTextInStyles** and **RecognizeInkWord** recognize ink text
  - **SetUserConfigEnMasse** sets a number of user configuration variables at once
  - **RegUserConfigChange** registers a function object to be called when a user configuration variable changes
  - **New transport methods**, **DeleteItem**, **DeleteRemoteItems**, and **RefreshOwner**, are related to remote items
  - **ROM\_GetSerialNumber** returns a Newton device's unique serial number
  - **SetScreenOrientation** changes the screen orientation
  - **TimeFrameStr** returns a string version of a date and time frame
- A number of existing functions and methods have been altered:
  - **view:DragAndDrop**
  - **BatteryStatus**
  - **extrasDrawer:GetPartEntryData**
  - **calendar:SetEntryAlarm**
  - **partFrame:ImportDisabled**
  - **LegalOrientations**
  - **GetAppParams**
  - **Gestalt**

## The Button Bar

---

Newton 2.1 OS introduces a software on-screen button bar.

## Miscellaneous

## The App Area

---

When you set your application's view bounds and justification, you are setting these values relative to the app area. The **app area** is defined as the rectangle relative to which your application's base view opens. That is, if you set your application's base view to full horizontal and vertical justification, your application covers exactly the app area. On some Newton devices the app area is the screen, on devices with a soft button bar, the app area is that portion of the screen not taken up by the soft button bar.

Your application does not have to exist entirely within the app area's bounds however, and in fact, it can exist entirely outside these bounds.











You can obtain information about the app area's position with `GetAppParams`. It now returns the extra slots `appAreaGlobalTop` and `appAreaGlobalLeft`, which hold the app area's top and left view bounds in global coordinates.

## Changing the Screen Orientation

---

The `GetOrientation` function has existed since Newton 2.0 OS, but was not previously documented. It returns an integer, as described in Figure 11-1, indicating the current screen orientation. Newton 2.1 OS provides the `SetScreenOrientation` function, to programmatically change the screen orientation. This function takes a single integer parameter, one of the screen orientation constants, specifying the desired screen orientation.

**Figure 11-1** Screen orientation constants

	MP 2000	eMate	MP 120/130
kPortrait 0			
kLandscape 1			NA
kPortraitFlip 2			NA
kLandscapeFlip 3			

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, the rotation fails. If some applications won't work in the new orientation, `SetScreenOrientation` may present the user with a dialog giving the option of canceling the rotation. If the user cancels the operation, the screen is not rotated, and `SetScreenOrientation` returns `nil`. You should always check the return value of `SetScreenOrientation`, or call `GetAppParams`, instead of assuming that the screen has been rotated.

The `LegalOrientations` function returns an array of all available screen orientations.

## Moving the Button Bar

The following attributes can be set programmatically:

- the position of the button bar on the screen.
- the placement of the controls (the up/down arrows and the overview button) within the button bar.

## Miscellaneous

- the position of the overview button in relation to the up/down arrows.

Each of these three attributes is controlled by a user configuration variable: `buttonBarPositions`, `buttonBarControlsPositions`, and `bellyButtonPositions`, respectively. Each of these variables contains a four-element array, one element for each screen orientation. These arrays are indexed by the screen orientation constants, see Figure 11-1 (page 11-4). That is, `array[kPortrait]` contains the value to use in the portrait orientation.

You should check for the existence of a software button bar before setting these values by testing if `GetRoot().buttons.soft` is non-`nil`. The following sample code sets the button bar to the left in either landscape orientation, and to the top in the two portrait orientations:

```
if GetRoot().buttons.soft then
    SetUserConfig('buttonBarPositions','[top,left,top,left]);
```

For information on accessing the user configuration data see “Functions for Accessing User Configuration Data” (page 19-58) in *Newton Programmer’s Guide*.

## Covering the “Soft” Button Bar

If you want to maximize the visible area of your application, you must cover the soft button bar if there is one. To do this, set your application base view’s bounds to a rectangle as large as the screen. Make sure to offset this rectangle to adjust for the fact that your application opens relative to the app area, not relative to the origin of the global coordinate system. The code example in Listing 11-1 demonstrates how to do this from within your application’s base view’s `ViewSetupFormScript`.

---

### Listing 11-1 Code to set an application’s view bounds to cover the entire screen

```
local params := GetAppParams();
self.viewbounds :=
    if GetRoot().buttons.soft then
        UnionRect(
            params.appAreaBounds,
            OffsetRect(
                params.buttonBarBounds,
                -params.appAreaGlobalLeft,
```

## Miscellaneous

```

        -params.appAreaGlobalTop));
    else
        params.appAreaBounds;

```

**Note**

While it is possible to get rid of the button bar to use the entire screen, it is not recommended; you should merely cover the button bar. ♦

## Disabling the “Silkscreened” Button Bar

Some Newton devices, such as the Message Pad 130, have the button bar off screen. On these devices the tablet is larger than the screen, to allow the user to tap on the button bar. If your goal is to disable these buttons, you should create a separate view, with the `BuildContext` function, that covers just these buttons. The following code shows how the bounds of such a view could be determined:

**Listing 11-2** Code to cover the silkscreened button bar

```

rb := GetRoot():LocalBox();
pb := GetAppParams().appAreaBounds;
self.viewbounds :=
    if rb.bottom = pb.bottom then // assume bb is on right
        SetBounds(pb.right, rb.top, rb.right, rb.bottom);
    else // assume bb is on bottom
        SetBounds(rb.left, pb.bottom, rb.right, rb.bottom);

```

**Note**

You could cover these buttons by creating a view that covers the entire root view. If you do this, you must not set the `vApplication` flag of that view, and this flag must be set all the way up the `_parent` chain, for the system to consider your view an application. For this reason it is recommended that you create a separate view to cover the silkscreened buttons. ♦

## Replacing the Button Bar

---

The `KillStdButtonBar` function closes the button bar, and reserves space for a replacement. The icons that were previously on the button bar are moved to the Extras Drawer. You should create a replacement for the button bar. Your replacement must allow the user to do the following:

- scroll up and down
- overview
- open the Extras Drawer

### ◆ WARNING

If you do not provide a replacement button bar with access to the Extras Drawer, you will have disabled the Newton device. ◆

Create your replacement button bar with the `BuildContext` function, making it a child of the root view. Your replacement button bar is not sent special system messages. It does not need to implement any of the functionality of the standard button bar, other than scrolling and overview support, and providing access to the Extras Drawer. And it does not need to register with the system in any way; specifically, you should not put a reference to your button bar in the `GetRoot().buttons.soft` slot.

You should register for changes in the “Packages” soup, with `RegSoupChange`, to be notified of changes such as a storage card being removed or a package being loaded. You should not, however, access the “Packages” soup. Simply rebuild your button bar when you are notified of a change of any type in this soup.

The `KillStdButtonBar` function accepts a single argument, a four-element array of frames, one for each possible screen orientation. The array is indexed by the screen orientation constants, that is, `array[kPortrait]` holds the frame for the portrait orientation. These frames should have two slots, `buttonBarPosition` and `buttonBarThickness`, specifying the location of the button bar in this orientation, and the amount of screen space to reserve for you button bar.

The following code sample reserves the bottom 20 pixels in all four screen orientations:

## Miscellaneous

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

Pass `nil` to `KillStdButtonBar` to restore the standard button bar.

**Note**

This function is intended to be used to replace the button bar. If you want to close the button bar so as to use that part of the screen for your application, you should merely cover the button bar, as described in “Covering the “Soft” Button Bar” (page 11-5). ♦

## Configuring the Button Bar

---

This section describes how to control which icons appear in the button bar, and in which order. This is something that should generally be left up to the user. It is easy enough for a user to drag an icon to and from the button bar.

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`. For a list of possible values for an icon's `labels` slot, see “Extras Drawer Folder Symbols” (page 11-17).

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; part entries are what the cursor returned by the Extras Drawer method `GetPartCursor` iterate over.

**Note**

The entries returned by the button bar method `GetPartEntries` are part entries, and as mentioned in the documentation for the Extras Drawer method `GetPartCursor`, you should not directly examine part entries. You can use the Extras Drawer `GetPartEntryData` method to retrieve information about a part entry. ♦



## Miscellaneous

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. In general, you should not make your application's icon fixed unless it is running on a Newton device dedicated for your application.

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 application symbols instead of part entries. This allows an icon to be added or removed from the Button Bar without having to look up 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 returns zero if the button bar is closed.

You can also change the position of the controls (the overview button and the up and down arrows), and the position of the arrows relative to the overview button, with two user configuration variables, `buttonBarControlsPositions` and `bellyButtonPositions`. These variables are defined in “New User Configuration Variables” (page 11-18).

## Changing the Spacing and Font of Icons

---

The distance, both horizontal and vertical, between icons in the button bar and the Extras Drawer is controlled by four user configuration variables: `buttonBarIconSpacingH`, `buttonBarIconSpacingV`, `extrasIconSpacingH`, and `extrasIconSpacingV`. You can also change the font used in both the button bar and the Extras Drawer with the user configuration variable `extraFont`.

These variables are defined in “New User Configuration Variables” (page 11-18).

### Note

Remember to allow space for multi-line icon titles when modifying the icon spacing. ♦

## The Clipboard

---

The underlying mechanism that supports the clipboard is drag and drop. The user selects some data object, and drags it off the screen onto the clipboard. The user can then drag and drop the clipboard contents into another view. As far as your application is concerned, it does not matter that the data is being dragged onto the clipboard instead of a view. Or conversely, that the data comes from the clipboard instead of another view. So, if your application supports the drag and drop API, it already allows the user to move items to and from the clipboard. For information on drag and drop, see “Dragging and Dropping with Views” (page 3-40) in *Newton Programmer’s Guide*.

There is also a direct interface to the clipboard with the `GetClipboard` and `SetClipboard` global functions. In some cases, you may wish to support cutting and pasting, but without the drag and drop interface. For example, it might not make sense to “select” data in your application. You can provide an alternate interface, such as a “Copy Data” button with the `GetClipboard` and `SetClipboard` functions.

If you do support the drag and drop API, you can, with very little work, also support the global editing command keys (Cmd-X for cut, Cmd-V for paste, and so on). See “Adding Global Editing Command Keys Support” (page 11-10).

### Adding Global Editing Command Keys Support

---

If your application already supports drag and drop, you can allow the user to cut and paste your data with the global command keys with minimal work. There is a new view slot `hilitedData` that you should set to `true` if there is data that can be cut or copied. You must also implement a new view method, `ViewAddDragInfoScript`. This method must create a *dragInfo* frame to be passed to your other view methods, such as the `ViewGetDropDataScript`. The `ViewAddDragInfoScript` method is needed, since the *dragInfo* frame would normally have been passed in the call to `DragAndDrop`.

## Miscellaneous

Supporting paste is automatic. Your normal drop reception methods are called: `ViewGetDropTypesScript`, `ViewDropScript`, `ViewDropDoneScript`, and so on.

## Using the Clipboard Functions

---

The `GetClipboard` and `SetClipboard` global functions provide a programmatic interface to the clipboard. They access the **clipboard frame**. This frame is returned by `GetClipboard`, and you should pass a frame such as this to `SetClipboard`. You may also pass `nil` to `SetClipboard` to clear the clipboard.

The following clipboard frame was obtained by dragging a meeting to the clipboard from the Dates application:

```
{
  label: "lunch at M...",
  types: [[meeting, text]],
  data: [[aMeetingFrame, aTextFrame]] ,
  bounds: {left: 15, top:86, right:75, bottom:118},
  bits: bitmapObject,
}
```

The `types` and `data` slots contains arrays with information about the clipboard items. Here these slots hold one-element arrays, since there is only one item. Each of these arrays itself contains a two element array. This indicates the different ways that the data can be interpreted. The ordering of these arrays is important, they are ordered by the preferred way to interpret the data. So in this example, it is best to interpret this data as a meeting, but this data can be pasted onto any view that can handle text. The `types` and `data` arrays must be synchronized, that is `types[i][j]` is the type of data in `data[i][j]`.

The following clipboard frame was obtained by cutting a section of a note containing a sketch and some text from the Notes application:

```
{
  label: "drawing",
  types: [[polygon], [text]],
  data: [[aPolygonFrame], [aTextFrame]] ,
  bounds: {left: 149, top:105, right:357, bottom:266},
  bits: bitmapObject,
}
```

## Miscellaneous

Note that there are two items in this clipboard frame, and that these two items can be interpreted in only one way: as a polygon and as text, respectively.

## Dial-In Networks

---

The dial-in network application programming interface (API) allows you to add dial-in networks to augment the built-in SprintNet and ConcertNet networks already in the system. A dial-in network provides phone numbers for an application (or transport) to call to get access to the network.

For example, a CompuServe mail client would need to register a CompuServe dial-in network to supply numbers for connecting to the CompuServe network.

The primary function of a dial-in network is to supply phone numbers to call given a particular location. It supplies these phone numbers by providing a function to be called by elements such as the connection slip and the Internet Enabler. This function returns the possible numbers.

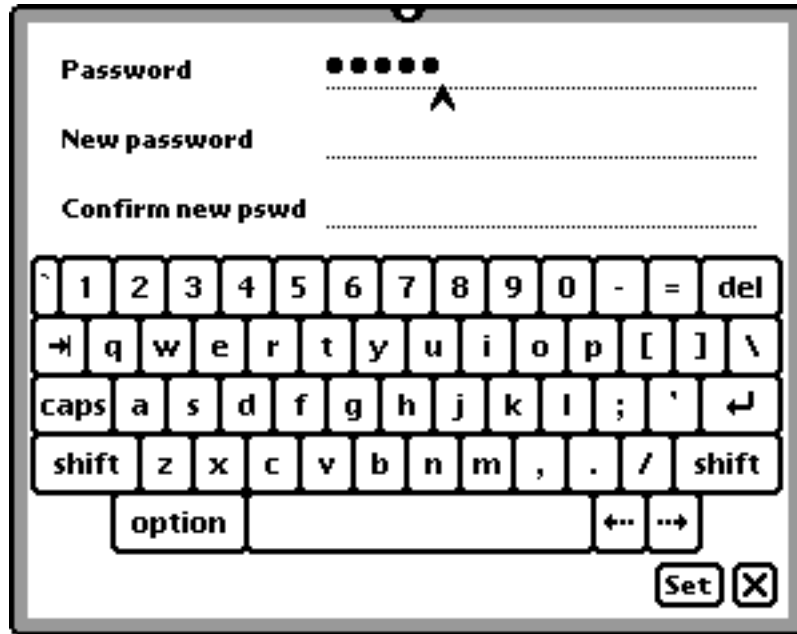
Dial-in networks are stored in a registry in the system. To register a dial-in network use the `RegDialInNetwork` function, passing in a network frame that describes the dial-in network; see “Dial-In Networks Network Frame” (page 11-21).

## Password Slip

---

The password slip provides a user interface element that allows users to enter and edit a password, it is available to you as `protoPasswordSlip`. The password slip uses blind entry lines, and these are also available to you, as `protoBlindEntryLine`, for use outside of password slips. A blind entry line is an entry line that echoes dummy characters to the user.

The password slip contains an embedded keyboard if displayed on a Newton device without a hardware keyboard, as show in Figure 11-2.

**Figure 11-2** A password slip

## Using protoPasswordSlip

The `protoPasswordSlip` handles much of the work of putting up a password slip. You are responsible only for implementing a scheme to store and retrieve (and optionally encrypt) the current password, and writing a function that handles the case of the correct password being entered. If the wrong password is entered, `protoPasswordSlip` informs the user of this.

When the user first enters a password, the password slip `SetPassword` method is called with a string argument. It is up to you to store this password in some way from within your `SetPassword` method, encrypting the string if security is an important factor. You must also write a `CurrentPassword` method to retrieve the password and return a string.

## Miscellaneous

The proto has a `MatchPassword` method that compares the string the user entered with the one you returned from `CurrentPassword`; override this method if you want something other than standard string comparison.

If the `MatchPassword` method returns non-`nil` (password matches), the password slip `MatchedPassword` method is called. You should override this method to perform any actions appropriate to the user having entered the correct password. Your override must then call `inherited:MatchedPassword` to close the slip.

The password slip can be used in three basic ways, as controlled by the `verifyPassword` slot. If it is set to `true`, the default, the user must enter a password, and may optionally change the password. If set to `nil`, the user is not prompted for the original password, but can change it. And if set to the symbol `'verifyOnly'`, the user is prompted for a password, but is not given the opportunity to change it.

## Using protoBlindEntryLine

---

The `protoBlindEntryLine` is used by `protoPasswordSlip`, but you can use it elsewhere to provide a private way for the user to enter text. This proto is based on `protoInputLine`. The only major difference is that this proto stores the actual text the user entered in a `realText` slot. The `text` slot contains a string of the same length as that in the `realText` slot, but using only the character specified by the `dummyChar` slot. By default, `dummyChar` is the bullet (•).

Aside from reading the `realText` slot instead of the `text` slot to see what the user entered, use this proto as you would `protoInputLine`.

## Transport Auto Put-Away Preference

---

There is a new transport configuration slot, `dontAutoPutAway`, that controls whether or not items received via that transport can be put away automatically. Set this slot to `'nil'` to allow items to be put away automatically. Set this slot to `'never'` to prevent items from being put away

## Miscellaneous

automatically. This slot is new in the Newton 2.1 OS. Here is an example of how to set this slot:

```
transport:SetConfig('dontAutoPutAway, 'never);
```

Note that in order for items to be put away automatically, the application to which the item is to be put away must also implement the `AutoPutAway` method in its base view. For more details on automatically putting away items, see “Automatically Putting Away Items” (page 21-31) in *Newton Programmer’s Guide*.

## Reference

---

### Data Structures

---

#### View Slot

---

The following view slot is new to Newton 2.1 OS:

##### Slot description

`hilitedData`

This slot states whether this view currently has data that can be cut or copied with the global command keys. If a view has this slot with a value of `true`, it is sent a `ViewAddDragInfoScript` message when the global command keys are used. If the command was a cut (as opposed to a copy) the view is also sent a `ViewDropRemoveScript` message.

### Clipboard Frame

---

A clipboard frame is the frame returned by `GetClipboard`, and passed to `SetClipboard`. It has the following slots:

## Miscellaneous

**Slot descriptions****label**

A string; the text displayed by the clipboard item.

**types**

Array of types arrays, one types array per item in the clipboard item. The number and order of these types arrays must match that of the data arrays in the `data` slot. Each types array contains symbols representing the types of data in the corresponding data array. Each symbol specifies the type of data in the corresponding element within the data array.

For example, the following 1-element `types` array describes a clipboard with one item, that can be seen as either text or as a picture:

```
'[ [text,picture] ]
```

Note that the nested array is ordered with the preferred type first. If the destination view accepts both text and pictures, the text is passed to the destination view.

This next 2-element `types` array on the other hand, describes a clipboard with two items, a string and a picture:

```
'[ [text],[picture] ]
```

The system can display types of `'text`, `'polygon`, `'ink`, and `'picture`. The type of data the system requires for these types is listed in Table 11-1.

**data**

Array of data arrays, one data array per item on the clipboard. The number and order of data arrays must match the number and order of types arrays in the `types` slot. Each data array should contain the data corresponding to that type in the array in the `types` slot. For example, the data in `clipboardFrame.data[i][j]` should be of the type specified by `clipboardFrame.types[i][j]`.

Each element within the nested arrays can be any NewtonScript object. If you specified a `'text`, `'polygon`,



Miscellaneous

bounds

'ink, or 'picture data type, these array elements should be frames with the slots listed in Table 11-1.  
A bounds frame; where the data came from in global coordinates.

Table 11-1     Clipboard data types accepted by the system

types	Required Slots	Optional slots
'text	text	any other c1ParagraphView slots
'polygon	points viewBounds	any other c1PolygonView slots
'ink	ink viewBounds	any other c1PolygonView slots
'picture	icon viewBounds	any other c1PictureView slots

Extras Drawer Folder Symbols

The following symbols are used for the labels slot of part entries by the Extras Drawer:

nil	Unfiled.
'_extensions	Extensions.
'_help	Help.
'_setup	Set up.
'_soups	Storage.
'_ButtonBar	The button bar.

Names Worksite Soup Entry

Worksite entries in the Names soup contain a cityAlias slot. The previous version of the Newton OS stored an entry alias to an undocumented soup in this slot. In Newton 2.1 OS this slot contains an array with information about the city, or nil if there is no city information. Note that ResolveEntryAlias returns nil if passed in an array (or anything other than a valid entry alias).

## Miscellaneous

## Newton Works Word Processor Soup Format

---

Newton Works word processor soup entries have the following slots.

### Slot descriptions

<code>class</code>	The symbol 'paper'.
<code>version</code>	Integer, the current version of the entry.
<code>title</code>	String which is the document title.
<code>timeStamp</code>	Creation date of the entry.
<code>realModTime</code>	Date the entry was most recently modified.
<code>saveData</code>	Frame returned from the <code>protoTXView</code> method <code>Externalize</code> (page 3-36).
<code>hiliteRange</code>	Frame with the document's highlight range (see "The Range Frame" (page 3-22)).
<code>margins</code>	Frame with slots <code>top</code> , <code>left</code> , <code>bottom</code> , <code>right</code> , which are the document's margins in pixels.

## New User Configuration Variables

---

The following user configuration variables are new to Newton 2.1 OS:

### Slot descriptions

<code>LCDContrast</code>	On units that support software control of the LCD contrast setting, this slot contains the current contrast setting. It can also be used to modify the current contrast. Use the <code>kGestaltArg_HasSoftContrast</code> Gestalt selector to check if a Newton device has software LCD control, and the maximum and minimum values.
<code>alarmVolumeDb</code>	Sets the system wide alarm volume in decibels. Use the <code>kGestaltArg_VolumeInfo</code> Gestalt selector to find the range of allowable values for the volume.
<code>soundVolumeDb</code>	Sets the system wide volume in decibels. Use the <code>kGestaltArg_VolumeInfo</code> Gestalt selector to find the range of allowable values for the volume.
<code>buttonBarPositions</code>	A 4-element array, specifying the position of the button bar in each of the four possible screen orientations. Each

## Miscellaneous

element in the array can be either `nil`, specifying that the default setting should be used, or one of the following symbols: `'top`, `'left`, `'right`, or `'bottom`.

The array elements are ordered using the screen orientation constants as indices to this array, see “Screen Orientation Constants” (page 11-27). That is, `buttonBarPositions[kPortrait]` should hold information for the portrait screen orientation.

`buttonBarControlsPositions`

A 4-element array, specifying the position of the controls—overview button and scroll arrows—in each of the four screen orientations. Each array element can be `nil`, specifying that the default value be used, or the symbols `'top` and `'bottom` for when the button bar is on the left or right sides of the screen, or `'left` and `'right` for when the button bar is on the top or bottom of the screen.

The array elements are ordered using the screen orientation constants as indices to this array, see “Screen Orientation Constants” (page 11-27). That is, `buttonBarControlsPositions[kPortrait]` should hold information for the portrait screen orientation.

`bellyButtonPositions`

A 4-element array, specifying the position of the overview button relative to the scroll arrows in each of the four screen orientations. Each array element can be `nil`, specifying that the default value be used, or the symbols `'outside`, `'inside`, `'left`, and `'right`.

The array elements are ordered using the screen orientation constants as indices to this array, see “Screen Orientation Constants” (page 11-27). That is, `bellyButtonPositions[kPortrait]` should hold information for the portrait screen orientation.

`buttonBarIconSpacingH`

An integer specifying the number of pixels between icons in the button bar when the button bar is laid out horizontally — across the top or bottom of the screen.

## Miscellaneous

The default is 40 on the MessagePad 2000. To restore this settings to its default value, set it to `nil`.

Check for the existence of a soft button bar; check if `GetRoot().buttons.soft` is non-`nil`, before setting this variable.

`buttonBarIconSpacingV`

An integer specifying the number of pixels between icons in the button bar when the button bar is laid out vertically — across the left or right sides of the screen. The default is 40 on the MessagePad 2000. To restore this settings to its default value, set it to `nil`.

Check for the existence of a soft button bar; check if `GetRoot().buttons.soft` is non-`nil`, before setting this variable.

`extrasIconSpacingH`

An integer specifying the horizontal spacing of icons in the Extras Drawer in pixels. The default is 64 in the MessagePad 2000. This value has no effect when the Extras Drawer is in overview mode. To restore this settings to its default value, set it to `nil`. This value is not used in systems prior to Newton 2.1 OS.

Check for the existence of a soft button bar; check if `GetRoot().buttons.soft` is non-`nil`, before setting this variable.

`extrasIconSpacingV`

An integer specifying the vertical spacing of icons in the Extras Drawer in pixels. The default is 52 in the MessagePad 2000. This value has no effect when the Extras Drawer is in overview mode. To restore this settings to its default value, set it to `nil`. This value is not used in systems prior to Newton 2.1 OS.

Check for the existence of a soft button bar; check if `GetRoot().buttons.soft` is non-`nil`, before setting this variable.

`extraFont`

The font used for the icon labels in both the Extras Drawer and the button bar. While you can use both an integer font spec or a font spec frame, it is strongly

## Miscellaneous

recommended that you use only integer font specs, such as `userFont9 + tsPlain` or `simpleFont9 + tsBold`. Using the integer representation in this instance accomplishes two things: it reduces NewtonScript Heap usage and it restricts you to the set of built-in fonts. Using a font that's not in ROM is extremely dangerous, because the font could be removed. This information is stored in a soup. A user may be forced to do a hard reset in order to remove a bad font specification.

This value is not used in systems prior to Newton 2.1 OS.

## Dial-In Networks Access Frame

---

An access frame contains the following slots:

### Slot descriptions

<code>mailNetwork</code>	A symbol for the network.
<code>mailPhone</code>	A string for the phone number.
<code>baud</code>	An integer indicating the baud rate.

## Dial-In Networks Network Frame

---

A network frame contains the following slots:

### Slot descriptions

<code>title</code>	A string describing the network, such as "SprintNet" or "ConcertNet".
<code>id</code>	A symbol uniquely identifying the network,
<code>GetAccessNumbers</code>	A function called to get access numbers for a worksite or city.

## Miscellaneous

**GetAccessNumbers**

---

*networkFrame*: `GetAccessNumbers(worksiteFrame, cityFrame)`

Called to retrieve an array of access numbers for a given worksite or city.

<i>worksiteFrame</i>	A frame of the format of a Names worksite soup entry; see “Worksite Entries” (page 16-22) in Chapter 16, “Built-in Applications and System Data Reference,” in <i>Newton Programmer’s Reference</i> .
<i>cityFrame</i>	A frame with the same format as the frames returned by the <code>GetCityEntry</code> function; see “GetCityEntry” (page 16-79) in <i>Newton Programmer’s Reference</i> .
return value	Return either an array of access frames, or <code>nil</code> if no numbers are available; access frames are described in “Dial-In Networks Access Frame” (page 11-21). You should never, however, return the empty array ( <code>[]</code> ).

**DISCUSSION**

It is up to you to implement a mechanism to store and retrieve these access numbers. One possible implementation is to store a frame containing this data in your package. If this data needs to be dynamic, to add new access numbers for example, you will probably want to create a soup for this data.

**Protos**

---

This section describes `protoPasswordSlip` and `protoBlindEntryLine`.

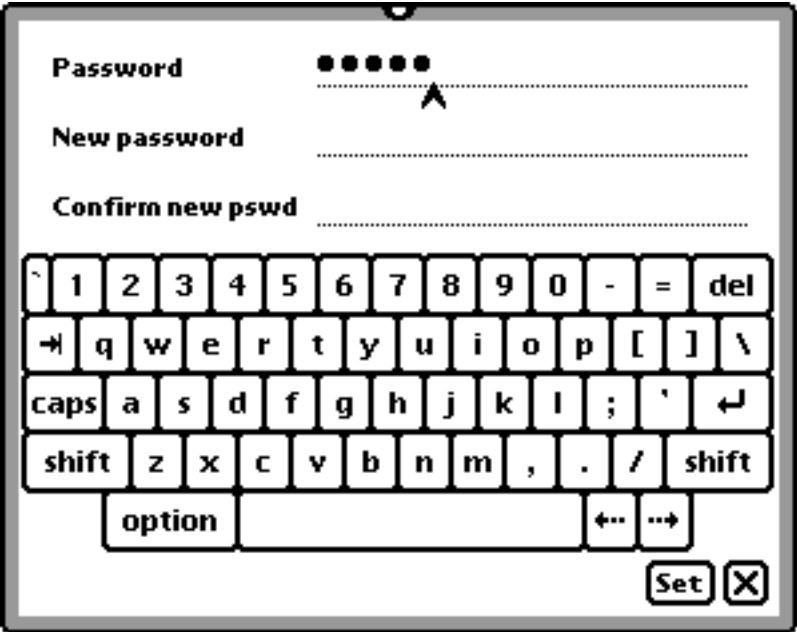
**protoPasswordSlip**

---

This proto allows the user to create a new password or enter an existing password without echoing the password in plain text. The typed keys appear as bullets in the input line. A view created from `protoPasswordSlip` is shown in Figure 11-3. Note that the slip does not include an embedded keyboard when created on a Newton device with a hardware keyboard attached.

Miscellaneous

**Figure 11-3** A view created from protoPasswordSlip



**Note**

This proto exists in Newton 2.0 OS, but was not previously documented. ♦

This proto has one slot of interest:

**Slot description**

`verifyPassword`

The symbol `'verifyOnly`, `true`, or `nil`. This slot determines if the password slip is used to just ask for a password, or if it is also used to change a password. A value of `'verifyOnly` specifies a password slip that queries a user for a password, but does not allow the

## Miscellaneous

user to change the password. In this case, the slip includes only a “Password” entry line.

A value of `true` means the user is queried for the old password, and may also change the password. This is the default. In this case the slip has all three entry lines: “Password,” “New Password,” and “Confirm Password.”

A value of `nil` means the user can change the password without entering the old one. In this case the slip includes only the “New Password” and “Confirm Password” entry lines.

This proto has the following methods of interest:

### CurrentPassword

---

*passwordSlip*: `CurrentPassword()`

Called to retrieve the current password.

return value      A string for the current password, or `nil` if there is no current password.

### DISCUSSION

You must supply this method. It is called when the proto needs to retrieve the current password in order to verify the entered password against it. If you stored the password in encrypted form, you should decrypt it before returning it.

### SetPassword

---

*passwordSlip*: `SetPassword(newPassword)`

Called to set a new password.

*newPassword*      A string, the new password to store.

return value      You can return anything; it is ignored.



## Miscellaneous

## DISCUSSION

You must supply this method. It is called when the user sets a new password, so that you can store it.

Note that the password is a string in plain text, so for maximum security you should encrypt it before storing it.

**MatchPassword**


---

*passwordSlip*:MatchPassword(*newPassword*, *currentPassword*)

Called to verify that the correct password has been entered

<i>newPassword</i>	A string for the password entered by user
<i>currentPassword</i>	A string for the current password as returned by <code>CurrentPassword</code> .
return value	Return <code>true</code> if the two match, <code>nil</code> if not.

## DISCUSSION

This method is supplied by `protoPasswordSlip`. You need to override it only if you want to compare the passwords using something other than a standard string comparison (`BinEqual` is used).

**MatchedPassword**


---

*passwordSlip*:MatchedPassword()

Called if a valid password was entered.

return value	You can return anything; it is ignored.
--------------	---

## DISCUSSION

This method is supplied by `protoPasswordSlip`. However, you should override it, supplying code that performs whatever actions you want as a result of the user entering the correct password.

You must call the inherited `MatchedPassword` method to correctly close the password slip.

## Miscellaneous

**protoBlindEntryLine**

This proto allows text to be entered, without echoing the text back to the user. This proto is used in the `protoPasswordSlip`. It is shown in Figure 11-4

**Figure 11-4** A view based on `protoBlindEntryLine`



This proto has three slots of interest:

**Slot descriptions**

<code>dummyChar</code>	Optional. A character containing the text to display instead of the real text. By default, the bullet character is used.
<code>realText</code>	The string the user has typed; set by <code>protoBlindEntryLine</code> . You should use this slot for looking up the value of the text (instead of looking in the <code>text</code> slot). Do not modify this slot directly. Use the <code>UpdateText</code> method.
<code>label</code>	Optional. The string used as the label of the entry line.

**UpdateText**

*blindEntryLine*:`UpdateText(newText)`

Sets the value of the `realText` slot to the value in *newText*, and correctly updates the dummy string displayed to the user.

<i>newText</i>	A string, the new value for the blind entry line.
return value	Undefined; do not rely on it.

Miscellaneous

DISCUSSION











This method is provided to give you the ability to programmatically set the value of the `realText` slot.

Constants

Screen Orientation Constants

The four screen orientation constants are shown in Figure 11-5.

**Figure 11-5** Screen orientation constants

	MP 2000	eMate	MP 120/130
kPortrait 0			
kLandscape 1			NA
kPortraitFlip 2			NA
kLandscapeFlip 3			

Serial Communication Tool Sound Option

There is a new serial communication tool option for enabling sound pass-through using a PCMCIA card. Here is an example of what the option looks like:

```
local option := {
```

## Miscellaneous

```

type: 'option,
label: kCMOPCMCIAModemSound, //"msnd"
opCode: opSetRequired,
form: 'template,
result: nil,
data: {
    arglist: [nil],
    typelist: ['struct', 'boolean'],
},
}

```

The `arglist` value is either `true` or `nil`. If `true`, sound pass-through is enabled. If `nil`, sound pass-through is disabled.

You would normally use this option with a PCMCIA modem using the serial tool. The modem tool automatically enables sound pass-through, so you should not need to use this option with the modem tool.

This option should be used only after the serial endpoint has connected.

**Note**

Sound pass-through should be disabled before the endpoint is disconnected. If it is not, power consumption increases and the speaker emits an annoying sound.

Sound pass-through only works for PCMCIA cards which support it through the PCMCIA specification.

This option is for use only in Newton OS 2.1 and higher. ♦

## Functions and Methods

---

The following methods and functions are either new to Newton 2.1 OS, have changed since previous OS releases, or have existed but were not previously documented. Unless otherwise noted in the COMPATIBILITY section of a function's description, all functions described here are new to Newton 2.1 OS.

## Views

---

The following functions are related to views.

## Miscellaneous

**DragAndDrop**

---

**view:** DragAndDrop(*unit*, *dragBounds*, *pinBounds*, *copy*, *dragInfo*)

Starts the drag and drop process, returning when the dragged item(s) is dropped into a view or into the clipboard; it is usually called from a ViewClickScript.

<i>unit</i>	The stroke unit received by the ViewClickScript method.						
<i>dragBounds</i>	The bounds of the item to be dragged, in global coordinates. The image enclosed by the bounds is used by the clipboard.						
<i>pinBounds</i>	A bounds frame or <i>nil</i> . The bounds to use when constraining the object within the app area. If you pass <i>nil</i> , the drag object's bounds, <i>dragBounds</i> , are used. If the object being dragged is almost the size of the app area, you may want to specify a smaller bounds frame than <i>dragBounds</i> , otherwise the object may not appear to move far enough. If you specify a bounds frame larger than <i>dragBounds</i> , the object cannot be dragged near the edge of the app area.						
<i>copy</i>	<i>Nil</i> or non- <i>nil</i> , indicating whether to drag a copy or the original items. Specify non- <i>nil</i> to drag a copy, <i>nil</i> to move the original items.						
<i>dragInfo</i>	An array of frames (one frame per dragged item). Each frame has the following slots: <table> <tr> <td><i>types</i></td><td>An array of symbols of the types to which an item can be converted.</td></tr> <tr> <td><i>dragRef</i></td><td>Any valid NewtonScript object. This value is passed to your other methods, such as your ViewGetDropDataScript.</td></tr> <tr> <td><i>label</i></td><td>An optional string used when the drop is to the clipboard; it is used as the clipboard label. If this slot is missing and the item</td></tr> </table>	<i>types</i>	An array of symbols of the types to which an item can be converted.	<i>dragRef</i>	Any valid NewtonScript object. This value is passed to your other methods, such as your ViewGetDropDataScript.	<i>label</i>	An optional string used when the drop is to the clipboard; it is used as the clipboard label. If this slot is missing and the item
<i>types</i>	An array of symbols of the types to which an item can be converted.						
<i>dragRef</i>	Any valid NewtonScript object. This value is passed to your other methods, such as your ViewGetDropDataScript.						
<i>label</i>	An optional string used when the drop is to the clipboard; it is used as the clipboard label. If this slot is missing and the item						

## Miscellaneous

has a `'text'` type, the text data is used as the label; otherwise a default label is used.

`minDragDistance`

An integer, the minimum distance in pixels that the user must drag the object before it moves. The default is 4.

return value

This method returns one of the following integers:

`kDragNot = 0` The item was not dragged at all.

`kDragged = 1` The item was dragged, but was rejected by the destination.

`kDragNDropped = 2`

The item was dropped into another view.

## DISCUSSION

The `DragAndDrop` method sends several messages to both the source view (the view from which `DragAndDrop` was sent) and the destination view (the view that will receive the items). If you want other views to be able to accept data, these views must implement all of the destination methods. If you have more than one view that can receive a drop, it is easier if you make one drop-aware proto and use it for your other views.

## SEE ALSO

For further information see “Dragging and Dropping with Views” (page 3-40) in *Newton Programmer’s Guide*.

See also “`DragAndDrop`” (page 1-4).

## COMPATIBILITY

The *dragInfo* argument’s `minDragDistance` slot is ignored in Newton operating systems prior to Newton 2.1.

## Miscellaneous

**DragAndDropLtd**

**view:** DragAndDropLtd(*unit*, *dragBounds*, *limitBounds*, *copy*, *dragInfo*)  
 //platform file function

Starts the drag and drop process, returning when the dragged item(s) is dropped into a view or into the clipboard; it is usually called from a ViewClickScript.

<i>unit</i>	The stroke unit received by the ViewClickScript method.				
<i>dragBounds</i>	The bounds of the item to be dragged, in global coordinates. The image enclosed by the bounds is used by the clipboard.				
<i>limitBounds</i>	<p>A bounds frame, or a frame with two optional slots: <i>limitBounds</i> and <i>pinBounds</i>. If you specify a bounds frame, it is the bounds in global coordinates in which the object can be dragged.</p> <p>Otherwise, you may pass in a frame with the following slots:</p> <table> <tr> <td><i>limitBounds</i></td><td>A bounds frame, the symbol 'none, or nil. The bounds frame is a rectangle in global coordinates in which the object can be dragged. The symbol 'none specifies that there is no limiting rectangle, and the object can be dragged anywhere on the screen. If you pass nil (or do not include a <i>limitBounds</i> slot) the app area is used as the limiting rectangle.</td></tr> <tr> <td><i>pinBounds</i></td><td>A bounds frame, the symbol 'none, or nil. The bounds to use when constraining the object within the limiting rectangle defined in the <i>limitBounds</i> slot. If you pass nil, the drag object's bounds, <i>dragBounds</i>, are used. If you pass 'none, an empty rectangle (with 0 width and height) is specified at the point where the pen went down to drag the object; that is, the</td></tr> </table>	<i>limitBounds</i>	A bounds frame, the symbol 'none, or nil. The bounds frame is a rectangle in global coordinates in which the object can be dragged. The symbol 'none specifies that there is no limiting rectangle, and the object can be dragged anywhere on the screen. If you pass nil (or do not include a <i>limitBounds</i> slot) the app area is used as the limiting rectangle.	<i>pinBounds</i>	A bounds frame, the symbol 'none, or nil. The bounds to use when constraining the object within the limiting rectangle defined in the <i>limitBounds</i> slot. If you pass nil, the drag object's bounds, <i>dragBounds</i> , are used. If you pass 'none, an empty rectangle (with 0 width and height) is specified at the point where the pen went down to drag the object; that is, the
<i>limitBounds</i>	A bounds frame, the symbol 'none, or nil. The bounds frame is a rectangle in global coordinates in which the object can be dragged. The symbol 'none specifies that there is no limiting rectangle, and the object can be dragged anywhere on the screen. If you pass nil (or do not include a <i>limitBounds</i> slot) the app area is used as the limiting rectangle.				
<i>pinBounds</i>	A bounds frame, the symbol 'none, or nil. The bounds to use when constraining the object within the limiting rectangle defined in the <i>limitBounds</i> slot. If you pass nil, the drag object's bounds, <i>dragBounds</i> , are used. If you pass 'none, an empty rectangle (with 0 width and height) is specified at the point where the pen went down to drag the object; that is, the				

## Miscellaneous

object moves until the tip of the pen reaches the limit bounds.

If the object being dragged is small, compared to the size of the `limitBounds`, you may want to specify a `pinBounds` smaller than *dragBounds*, otherwise the object may not appear to move far enough. If you specify a bounds frame larger than *dragBounds*, the object cannot be dragged near the edge of the `limitBounds`.

*copy*

`Nil` or `non-nil`, indicating whether to drag a copy or the original items. Specify `non-nil` to drag a copy, or `nil` to move the original items.

*dragInfo*

An array of frames (one frame per dragged item). Each frame has the following slots:

<code>types</code>	An array of symbols of the types to which an item can be converted.
<code>dragRef</code>	Any valid NewtonScript object. This value is passed to your other methods, such as your <code>ViewGetDropDataScript</code> .
<code>label</code>	An optional string used when the drop is to the clipboard; it is used as the clipboard label. If this slot is missing and the item has a <code>'text</code> type, the text data is used as the label; otherwise a default label is used.
<code>minDragDistance</code>	An integer, the minimum distance in



## Miscellaneous

pixels that the user must drag the object before it moves. The default is 4.

## return value

This method returns one of the following integers:

`kDragNot = 0` The item was not dragged at all.

`kDragged = 1` The item was dragged, but was rejected by the destination.

`kDragNDropped = 2`

The item was dropped into another view.

**IMPORTANT**

This function is not defined in all ROM versions and is supplied by the NTK Platform file. This implementation, as a global function, and not as a view method, requires an additional argument *view*, the view calling this function.

Call it using this syntax:

```
call kDragAndDropLtdFunc with (view, unit, dragBounds, limitBounds,  
copy, dragInfo));
```

**DISCUSSION**

The `DragAndDropLtd` method sends several messages to both the source view (the view from which `DragAndDropLtd` was sent) and the destination view (the view that will receive the items). If you want other views to be able to accept data, these views must implement all of the destination methods. If you have more than one view that can receive a drop, it is easier if you make one drop-aware proto and use it for your other views.

**SEE ALSO**

This function is discussed in “DragAndDrop” (page 1-4) in Chapter 1, “NPG/NPR 2.0 Errata.”

## Miscellaneous

**ViewAddDragInfoScript**

---

**view:** ViewAddDragInfoScript(*dragInfo*)

Called to retrieve data if the user presses the global command keys when your view is the key view, and your view has a `hilitedData` slot set to `true`.

<i>dragInfo</i>	An array of frames. You should add a frame to this array if you have something to cut or copy. Your frame should have the following slots:
<code>types</code>	An array of symbols of the types to which an item can be converted.
<code>view</code>	A view object type if the dragged item is a view with a symbol type of 'paragraph', 'polygon', 'picture, and so on.
<code>dragRef</code>	Any value that will be passed to other methods, such as the <code>ViewGetDropDataScript</code> .
<code>label</code>	An optional string used when the drop is to the clipboard; it is used as the clipboard label. If this slot is missing and the item has a 'text' type, the text data is used as the label; otherwise a default label is used.
<code>minDragDistance</code>	An integer, the minimum distance in pixels that the user must drag the object before it moves. The default is 4.
return value	Return <code>true</code> if you have added an element to <i>dragInfo</i> ; that is, something was cut or copied. Return <code>nil</code> otherwise.

**Stationery**

---

The following functions are related to stationery.

## Miscellaneous

**RegStationeryChange**

---

RegStationeryChange(*regSymbol*, *functionBody*)

Registers a function object to be executed when stationery is installed or removed.

<i>regSymbol</i>	A unique symbol that includes your developer signature.
<i>functionBody</i>	Function object called when stationery changes. This function body takes four arguments:
<i>message</i>	A symbol; currently the symbols 'install and 'remove are used.
<i>defType</i>	A symbol; currently the symbols 'dataDef and 'viewDef are sent, for the type of stationery that has been installed or removed.
<i>symbol1</i>	The dataDef symbol of the installed or removed stationery.
<i>symbol2</i>	If <i>defType</i> is 'dataDef, then this is undefined. If <i>defType</i> is 'viewDef, then this is the viewDef symbol of the installed or removed stationery.
return value	Undefined; do not rely on it.

**SPECIAL CONSIDERATIONS**

The function passed in the *functionBody* argument must not itself call RegStationeryChange or UnregStationeryChange.

**UnRegStationeryChange**

---

UnRegStationeryChange(*regSymbol*)

Unregisters a function body previously registered using RegStationeryChange.

<i>regSymbol</i>	The symbol used in the call to RegStationeryChange.
return value	Undefined; do not rely on it.

## Miscellaneous

## Text

---

The following function is related to text handling.

**MakeFontMenu**


---

MakeFontMenu(*font*, *families*, *sizes*, *styles*)

Creates an array of font menu items, including font families, sizes, and styles.

*font* Nil or a font specification as either a frame or a packed integer that represents the current font. In the returned font menu, the family, size, and style of this font is check-marked to indicate that it is the current font. Passing nil results in no items being check-marked. You can also pass an array of font specs. In this case, any family, size, or style common to all font specs will be check-marked. Also, the size choices will be a union of the possible sizes for each of the fonts in the array.

*families* Nil, the symbols 'all or 'none, or an array of font families. This parameter controls which fonts are returned. If this parameter is nil, all user fonts in the system are included in the menu. If you pass the symbol 'all, every font is included, even the system font. If you pass the symbol 'none, font families are not included in the returned menu. An array specifies the list of font families to include in the menu.

**Note**

Some Newton devices contain undocumented fonts. For example the eMate 300 includes the Courier font. ♦

*sizes* Nil, the symbol 'none, or an array of numbers. This parameter controls which font sizes are included. If you pass nil, the font size specified in the *font* parameter is used. If you pass the symbol 'none, font size choices are not included in the menu. An array specifies the list of sizes to include in the menu.

*styles* Nil, the symbol 'none, or an integer. This parameter controls which font style choices are included. If you

Miscellaneous

	pass <code>nil</code> , the default styles in the system are included. If you pass the symbol <code>'none</code> , style choices are not included in the menu. An integer specifies a list of style choices to return for the menu, encoded as a packed integer; the style constants are listed in the discussion section. To specify more than one font face constant, simply add them together, and pass in the sum.
return value	An array of font menu items, suitable for use wherever a pop up menu array is needed, such as in <code>protoPopupButton</code> , <code>protoPopInPlace</code> , and the <code>PopupMenu</code> view method.

DISCUSSION

Use these constants to specify the font style in NewtonScript font frames.

Constant	Value
<code>kFaceNormal</code>	<code>0x000</code>
<code>kFaceBold</code>	<code>0x001</code>
<code>kFaceItalic</code>	<code>0x002</code>
<code>kFaceUnderline</code>	<code>0x004</code>
<code>kFaceOutline</code>	<code>0x008</code>

Presently, the *styles* argument ignores the constants `kFaceSuperscript` and `kFaceSubscript`, which are otherwise valid font style constants.

Recognition

---

The following functions are related to recognition.

## Miscellaneous

**RecognizeTextInStyles**

---

`RecognizeTextInStyles(textFrame, defaultFontSpec)`

Translates the ink words in a frame containing a combination of raw ink and text.

<i>textFrame</i>	A frame with a <code>text</code> and a <code>styles</code> slot.
<i>defaultFontSpec</i>	A font spec, either an integer or a frame. This font is used for translated ink. For more information on font specs, see Chapter 8, “Text and Ink Input and Display,” in <i>Newton Programmer’s Guide</i> .
return value	If <i>textFrame</i> contains no ink, <i>textFrame</i> is returned. Otherwise a new frame is returned. This frame has a <code>text</code> and a <code>styles</code> slot, containing translated versions of all the ink words.

**DISCUSSION**

The highest confidence match for each ink word is returned.

**RecognizeInkWord**

---

`RecognizeInkWord(inkWord)`

Returns an array of translation options for an ink word.

<i>inkWord</i>	Ink word data from a rich string or from a style array.
return value	An array of frames for each possible match, or <code>nil</code> if no matches were found. The frames in the array contain a <code>word</code> slot which contains a string.

**DISCUSSION**

The array returned is sorted such that higher confidence matches are earlier in the array; that is the first element is the highest confidence match.

**System Services**

---

The following functions are related to system services.

## Miscellaneous

**BatteryStatus**

---

`BatteryStatus(which)`

Returns a status frame for the specified battery.

*which*                      An integer identifying the battery for which to return status information. The value 0 specifies the primary battery pack.

return value              A status frame; see DISCUSSION.

**DISCUSSION**

The status frame returned contains the following slots:

<code>batteryType</code>	Contains one of the following symbols, or an integer: <table> <tr> <td>'alkaline</td><td>Battery is standard alkaline.</td></tr> <tr> <td>'nicd</td><td>Battery is nickel-cadmium.</td></tr> <tr> <td>'nimh</td><td>Battery is nickel-metal hydride.</td></tr> <tr> <td>'lithium</td><td>Battery is lithium.</td></tr> </table>	'alkaline	Battery is standard alkaline.	'nicd	Battery is nickel-cadmium.	'nimh	Battery is nickel-metal hydride.	'lithium	Battery is lithium.
'alkaline	Battery is standard alkaline.								
'nicd	Battery is nickel-cadmium.								
'nimh	Battery is nickel-metal hydride.								
'lithium	Battery is lithium.								
<code>batteryVoltage</code>	A real number giving the current battery voltage.								
<code>batteryCapacity</code>	An integer, indicating the percentage of a full charge that the battery contains.								
<code>batteryLow</code>	An integer, indicating the percentage of a full charge at which the “low battery” warning should be triggered by the system.								
<code>batteryDead</code>	An integer, indicating the percentage of a full charge at which the “dead battery” warning should be triggered and the unit shut down by the system.								
<code>batteryCurrent</code>	A real number indicating the current drain, in milliamps. This slot is <code>nil</code> if the battery is charging. This slot is new in 2.1.								
<code>acPower</code>	Contains a symbol ('yes or 'no) indicating whether or not the unit has AC power applied. Note that this does								

## Miscellaneous

	not imply that the battery is charging. See <code>chargeState</code> to determine that.
<code>acVoltage</code>	A real number giving the AC voltage being supplied by an AC adapter, or <code>nil</code> if AC power is not supplied.
<code>chargeState</code>	Contains one of the following symbols, or an integer: <ul style="list-style-type: none"> <li><code>'notCharging</code> The battery is not charging.</li> <li><code>'discharging</code> The battery is discharging.</li> <li><code>'preliminaryCharging</code> The battery is charging under a pulsed duty schedule that raises its voltage to a level at which it can be efficiently fast-charged. This charging mode is used initially for charging a heavily discharged battery.</li> <li><code>'fastCharging</code> The battery is fast-charging.</li> <li><code>'trickleChargeContinuous</code> or <code>'trickleCharging</code> The battery is fully charged and is being maintained in that state by trickle-charging.</li> </ul>
<code>chargeRate</code>	Reserved for future use.
<code>chargeCurrent</code>	A real number indicating the current, in milliamps, being supplied to charge the battery, if it is charging. If the battery is discharging, this is the current supplied from the battery to the system.
<code>ambientTemp</code>	A real number indicating the ambient temperature in degrees Celsius.
<code>batteryTemp</code>	A real number indicating the battery temperature in degrees Celsius.



Miscellaneous

**Note**

A `nil` value for a slot means the underlying hardware cannot supply this information. The slots containing symbol values (`batteryType`, `chargeState`, `acPower`) may contain integers if the battery driver returned something other than the values listed here. ♦

**COMPATIBILITY**

The return value of this function is changed from its Newton 2.0 OS implementation. The `batteryCurrent` slot is new and the possible symbol values for the `chargeState` slot are different.

**Built-in Applications and System Data**

---

The following functions are related to built-in applications and system data.

**GetPartEntryData**

---

*extrasDrawer*:GetPartEntryData(*entry*) //platform file function

Returns a frame containing information about an Extras Drawer part entry.

<i>entry</i>	An entry obtained from a part cursor; by using <code>GetPartCursor</code> .	
return value	The frame returned has the following slots:	
	<code>icon</code>	A bitmap object, containing the bitmap for the part icon displayed in the Extras Drawer on Newton 1.x and 2.0 operating systems.
	<code>iconPro</code>	A frame containing two pix families, for the highlighted icon and the normal icon to display in the Extras Drawer on Newton 2.1 OS. For more information on

## Miscellaneous

	gray icons and pix families, see Chapter 6, “Drawing and Graphics 2.1.”
<code>text</code>	A string that is the text shown under the part icon.
<code>labels</code>	A symbol identifying the Extras Drawer folder in which the part is filed. For a list of these see “Extras Drawer Folder Symbols” (page 11-17).
<code>appSymbol</code>	A symbol identifying the application, if the part frame has an <code>app</code> slot.
<code>packageName</code>	A string that is the name of the package that contains the part.

**COMPATIBILITY**

The return value of this function is changed from its Newton 2.0 OS implementation. The `iconPro` slot is new for the Newton 2.1 OS.

**IMPORTANT**

This function is not defined in all ROM versions and is supplied by the NTK Platform file. Call it using this syntax:

```
call kGetPartEntryDataFunc with (entry);
```

Note that this function is implemented in ROM on Newton 2.1 units, so you can call it directly if your application runs only on the Newton 2.1 OS. ♦

## Miscellaneous

**SetEntryAlarm**

---

*calendar*: SetEntryAlarm(*mtgText*, *mtgStartDate*, *minutesOrDaysBefore*)

Sets an alarm for the meeting or event with the given text at the given date and time. If the meeting or event is an instance of a repeating meeting or event, the alarm is set for all instances of the repeating meeting or event.

<i>mtgText</i>	A string or rich string that is the meeting text of the meeting or event for which you want to set the alarm time.
<i>mtgStartDate</i>	An integer specifying the start date and time of the meeting or event, in the number of minutes passed since midnight, January 1, 1904.
<i>minutesBefore</i>	A non-negative integer, which specifies how far in advance of the meeting or event the alarm should go off. A value of 0 means the alarm goes off at the time of the meeting. This integer should specify the number of minutes before <i>mtgStartDate</i> that you want the alarm to go off for a meeting, and the number of days before <i>mtgStartDate</i> for an event.  You can specify <code>nil</code> to clear an alarm that is currently set.
return value	Undefined; do not rely on it.

**COMPATIBILITY**

The version of this function available on Newton 2.0 OS can only be used for meetings. The `kSetEventAlarmFunc` function exists in the 2.0 platform file to set alarms for events.

**SetUserConfigEnMasse**

---

SetUserConfigEnMasse(*changeSym*, *changeFrame*)

Sets one or more user configuration variables and broadcasts changes.

<i>changeSym</i>	A symbol passed to functions registered for notification of user configuration changes. This symbol should be one of the slot names in <i>changeFrame</i> . Some functions
------------------	--

## Miscellaneous

	registered for user configuration variable changes are passed only this symbol, see <code>RegUserConfigChange</code> .
<i>changeFrame</i>	A frame where the names of the slots are the names of the user configuration variables that you wish to set, and the slot values are the values to which the respective user configuration variables should be set.
return value	Undefined; do not rely on it.

## DISCUSSION

Changes are broadcasted to functions registered via the `RegUserConfigChange` function.

**RegUserConfigChange**


---

`RegUserConfigChange(callbackID, callbackFn)`

Registers a function object to be called each time a user configuration variable changes.

<i>callbackID</i>	A unique symbol identifying the function object to be registered; normally, the value of this parameter is the application symbol, which includes your registered signature, or some variation on it.
<i>callbackFn</i>	<p>A function object called when a user configuration variable changes. It is passed either one or two parameters. This function can be of either of the following two forms:</p> <pre>func(<i>changeSym</i>, <i>changeFrame</i>) begin .... end</pre> <pre>func(<i>changeSym</i>) begin .... end</pre> <p>On Newton devices where the <code>SetUserConfigEnMasse</code> function is not defined, this callback function is always passed one argument. On Newton devices with <code>SetUserConfigEnMasse</code> defined, this function will be called with the proper number of arguments; that is, if you define a one argument function, it will be called with only the <i>changeSym</i> argument, but if you define it</p>

## Miscellaneous

with two arguments, it is called with both the *changeSym* and the *changeFrame* arguments.

For information on the *changeSym* and the *changeFrame* parameters, see `SetUserConfigEnMasse`.

The return value of *callBackFn* function is ignored.

return value      Undefined; do not rely on it.

## DISCUSSION

Note that it is up to the application that changed one of these variables to broadcast the change. This is not something that you need to worry about, since the `SetUserConfig` and `SetUserConfigEnMasse` functions always broadcast the changes. Also note that the system may change, and broadcast the change of, certain undocumented user configuration variables; you should ignore these symbols.

## SPECIAL CONSIDERATIONS

The function *callBackFn* must not call the `RegUserConfigChange` or `UnRegUserConfigChange` functions.

## COMPATIBITLY

This function exists in Newton 2.0 OS.

**KillStdButtonBar**


---

`KillStdButtonBar(buttonBarParams)`

Closes (or restores) the standard button bar, and reserves screen area for a new one.

*buttonBarParams*      A 4-element array or `nil`. Pass the value `nil` to restore the standard button bar. If you pass an array, each element should be a frame specifying where to save screen space for the replacement button bar in the four different screen orientations. The array elements should be ordered as specified by “Screen Orientation Constants” (page 11-27); for example,

## Miscellaneous

*buttonBarParams*[*kPortrait*] should hold information for the portrait screen orientation.

These frames should have the following slots:

*buttonBarPosition*

Required. One of the following symbols:

'top', 'bottom', 'right', 'left, or 'none.

These symbols specify where to reserve space for the replacement button bar.

Specify 'none if you do not wish to reserve this space.

*buttonBarThickness*

An integer specifying how much space to save for the button bar in pixels. You may not omit this slot, unless

*buttonBarPosition* is set to 'none.

return value      Undefined; do not rely on it.

## DISCUSSION

If the application area becomes less than 320 pixels high as a result of a call to *KillStdButtonBar*, views without a *ReorientToScreen* method cannot open.

**GetPartEntries**


---

*buttonBar*: *GetPartEntries*()

Returns the part entries of all icons in the button bar.

return value      A frame with the following two slots, *fixed* and *mobile*. Both of these slots contain an array of part entries. The icons of the part entries in *fixed* cannot be moved by dragging. Similarly, the icons of the part entries in *mobile* can be moved. The ordering of these arrays is important; it determines the order of the icons in the button bar.

## Miscellaneous

## DISCUSSION

You must not modify the part entries in any way. To obtain information from a part entry, use the Extras Drawer `GetPartEntryData` method.

To send the `GetPartEntries` method, use code such as the following:

```
local bb := GetRoot().Buttons;
if (bb.soft) then bb:GetPartEntries();
```

**Reconfigure**


---

*buttonBar*: `Reconfigure(newSetup)`

Reconfigures the button bar.

*newSetup*                      A frame with *fixed* and *mobile* slots. Each slot should contain an array of part entries or application symbols. The icons represented by part entries in *fixed* are not draggable, while the ones in *mobile* are. The ordering of these arrays is important; it determines the order of the icons in the button bar.

return value                  Undefined; do not rely on it.

## DISCUSSION

To send this method, use code such as the following:

```
local bb := GetRoot().Buttons;
if (bb.soft) then bb:Reconfigure(newSetup);
```

**IconCapacity**


---

*buttonBar*: `IconCapacity()`

Returns the number of icons the button bar can currently hold.

return value                  An integer, the maximum number of icons.

## DISCUSSION

To send this method use code such as the following:

```
local bb := GetRoot().Buttons;
if (bb.soft) then bb:IconCapacity();
```

## Miscellaneous

Transports

---

The following functions are related to transports.

**DeleteItem**

---

*transport*:DeleteItem(*item*)

Deletes an item from the In/Out Box.

*item*                      The item to delete. This is an item frame from the In Box.

return value              Undefined; do not rely on it.

**DeleteRemoteItems**

---

*transport*:DeleteRemoteItems()

Causes the transport to delete from the In/Out Box all remote items that have not been fully downloaded.

return value              Undefined; do not rely on it.

**DISCUSSION**

Typically, you use the DeleteRemoteItems method after the transport disconnects, to remove from the In/Out Box all remote items that the user chose not to retrieve fully. This method removes all items whose remote slot is set to true.

**COMPATIBILITY**

This 2.1 method replaces the 2.0 method *ownerApp*:RemoveTempItems. If you are writing an application for 2.1 only, then you should use this method instead of *ownerApp*:RemoveTempItems.

**RefreshOwner**

---

*transport*:RefreshOwner()

Causes the transport owner (typically the In/Out Box) to refresh the view of the in box.

return value              Undefined; do not rely on it.



## Miscellaneous

## DISCUSSION

You use `RefreshOwner` to refresh the in box view after remote items are fully retrieved and after remote items that are not fully retrieved are deleted.

## COMPATIBILITY

This 2.1 transport method replaces the 2.0 method `ownerApp.Refresh`. If you are writing an application for 2.1 only, then you should use this method instead of `ownerApp.Refresh`.

## Dial-In Networks

The following functions are related to dial-in network support.

**RegDialinNetwork**

`RegDialinNetwork(networkSym, networkFrame)`

Registers a new dial-in network with the system.

<i>networkSym</i>	A symbol uniquely identifying the network
<i>networkFrame</i>	A network frame, as described in “Dial-In Networks Network Frame” (page 11-21).
return value	Undefined; do not rely on it.

## DISCUSSION

This function should usually be called from your part’s `InstallScript`, as in the following code sample:

```
DefineGlobalConstant ( 'dudeNetFrame,
{
    title: "DudeNet",
    id: 'dudeNet,
    GetAccessNumbers: func(worksite,city)
        begin
            local result := [];
            if worksite then
                AddArraySlot (
                    result,
                    {
                        mailPhone:"111-1111",
```

## Miscellaneous

```

        mailNetwork: 'dudeNet',
        baud: 9600
    }
)
if city then
    AddArraySlot (
        result,
        {
            mailPhone:"222-2222",
            mailNetwork: 'dudeNet',
            baud: 2400
        }
    )
end
result;
end
}
);
partData := {};
InstallScript := func(partFrame,removeFrame) //auto part
begin
    call kRegDialinNetworkFunc with ('dudeNet,dudeNetFrame);
end;

```

**UnRegDialinNetwork**

---

UnRegDialinNetwork(*networkSym*)

Unregisters a dial-in network that had been registered with a call to RegDialinNetwork.

<i>networkSym</i>	The symbol used in the call to RegDialinNetwork.
return value	Undefined; do not rely on it.

**DISCUSSION**

This function should usually be called from your part's RemoveScript.

## Miscellaneous

**GetLocAccessNums**

---

`GetLocAccessNums(entry, which)`

Retrieves an array of access frames given a location frame and an array of dial-in network symbols to look for.

**entry** A location frame. Can be a worksite or a city location. If `nil`, `GetLocAccessNums` uses the current emporium and city location.

For information on these various entities see the following sections of Chapter 16, “Built-in Applications and System Data Reference,” in *Newton Programmer’s Reference* :

**worksites** “Worksite Entries” (page 16-22), worksite entries are a type of Names soup entry.

**cities** “GetCityEntry” (page 16-79), the `GetCityEntry` function returns a city location frame.

**the current emporium** “User Configuration Variables” (page 16-101), the `currentEmporium` variable contains an alias to a Names worksite soup entry.

**which** An array of network symbols. Usually the transport's `networkSymbols` array if the Mail Enabler is used. Matches to all these symbols are returned.

**return value** Returns an array of access frames; see “Dial-In Networks Access Frame” (page 11-21).

**Note**

If the mail transport does not contain the *networkSym* for the dial-in network within its `networkSymbols` slot, the network phone numbers will not appear in the connection slip. ♦

## Miscellaneous

**GetAllDialinNetworks**

---

`GetAllDialinNetworks()`

Returns an array of all the dial-in network frames registered in the system.

return value      An array of network frames, see “Dial-In Networks Network Frame” (page 11-21).

**GetDialinNetwork**

---

`GetDialinNetwork(networkSym)`

Returns the dial-in network frame that corresponds to *networkSym*.

*networkSym*      The symbol of the network whose frame to return.  
return value      A network frame; see “Dial-In Networks Network Frame” (page 11-21).

**Utility Functions**

---

The following functions are miscellaneous utility functions.

**GetClipboard**

---

`GetClipboard()`

Returns the contents of the clipboard.

return value      A clipboard data frame, or `nil` if the clipboard is empty. Clipboard data frames are described in “Clipboard Frame” (page 11-15).

**SetClipboard**

---

`SetClipboard(clipboardData)`

Sets the contents of the clipboard.

*clipboardData*      A clipboard data frame, as described in “Clipboard Frame” (page 11-15), or `nil` to clear the clipboard. In

## Miscellaneous

addition to the slots in a normal clipboard data frame, you may include an `xy` slot in *clipboardData*:

`xy`                    A frame with two slots `x` and `y`. Each slot contains an integer specifying the offset from the origin, in global coordinates, of the label's position on the screen. By default, the clipboard label is placed on the left side of the screen, a little below the top.

return value            Undefined; do not rely on it.

**DISCUSSION**

You can use this function to perform a paste. Use `GetClipboard` to get the contents, then call `SetClipboard` with `nil` to clear the clipboard.

**ROM\_GetSerialNumber**


---

```
ROM_GetSerialNumber()
```

Returns the unique hardware serial number of a Newton device.

return value            An 8 byte binary object containing the Newton device's serial number.

**DISCUSSION**

This function is not defined in neither Newton 1.x nor 2.0 OS. You should wrap the call to this function in a `try...onException` block, as in the following example:

```
local sn;

try
    sn := call ROM_GetSerialNumber with ()
onException |evt.ex| do
    nil;

if sn then
    // ...
```

## Miscellaneous

The serial number returned in ROM is not the same as the serial number stamped on the Newton device. The ROM serial number is intended for use by programmers.

The `StrHexDump` and `ExtractByte` functions are designed to read binary objects.

**ImportDisabled**


---

*partFrame*: `ImportDisabled(unitName, majorVersion, minorVersion)`

Called after an imported unit has been deactivated to perform housekeeping.

<i>unitName</i>	A symbol, the name of the unit.
<i>majorVersion</i>	An integer, the major version number of the unit.
<i>minorVersion</i>	An integer, the minor version number of the unit.
return value	Either the symbol <code>'ThrillMeChillMeFulfillMe'</code> or anything else.

**DISCUSSION**

The part should deal with the situation as gracefully as possible. For example, you could use alternative data, or put up a message slip with the `Notify` method and/or close your application.

If you return the symbol `'ThrillMeChillMeFulfillMe'`, the system attempts to re-resolve the imports. For example, if version 2 of unit `foo` is disabled and your package's `ImportDisabled` script returns `'ThrillMeChillMeFulfillMe'`, the system looks for other versions of the objects in the unit `foo`.

**COMPATIBILITY**

Newton 2.0 OS sends this message, but ignores the return value.

**LegalOrientations**


---

`LegalOrientations()`

Returns the legal values for screen orientations on the Newton device.

return value	An array of integers; possible values are listed in “Screen Orientation Constants” (page 11-27).
--------------	--

## Miscellaneous

## COMPATIBILITY

This function is supported in Newton OS 2.0. On the MessagePad 120 and 130 units, the only possible return values are `kPortrait` (0) and `kLandscapeFlip` (3).

**SetScreenOrientation**

---

`SetScreenOrientation(orientation)`

Sets the screen orientation.

<i>orientation</i>	An integer specifying the new orientation; possible values are listed in “Screen Orientation Constants” (page 11-27).
return value	<code>Nil</code> if the screen orientation was not changed, otherwise a non- <code>nil</code> value is returned.

## DISCUSSION

This function requests the system to rotate the screen to the desired orientation. The user may be prompted if particular applications do not support the new orientation.

**GetAppParams**

---

`GetAppParams()`

Returns a frame containing information about the screen size and other system configuration items.

return value	A frame with the following slots:
<code>appAreaTop</code>	The y coordinate of the top-left corner of the application area. Children of the root

## Miscellaneous

	view are always opened relative to the application area. This value is always 0.
<code>appAreaLeft</code>	The x coordinate of the top-left corner of the application area. This value is always 0.
<code>appAreaWidth</code>	The width of the screen in pixels.
<code>appAreaHeight</code>	The height of the screen in pixels.
<code>buttonBarPosition</code>	A symbol, either 'top', 'left', 'bottom', 'right, or 'none indicating where the button bar is, if there is one. This is useful if you want to locate your application flush against the button bar.
<code>appAreaGlobalTop</code>	The y coordinate of the top of the application area in global coordinates.
<code>appAreaGlobalLeft</code>	The x coordinate of the left of the application area in global coordinates.
<code>buttonBarBounds</code>	If there is a soft button bar this slot contains its view bounds.

## COMPATIBILITY

Versions of this function previous to Newton 2.1 OS return a frame without the `appAreaGlobalTop`, `appAreaGlobalLeft`, and `buttonBarBounds` slots.

**Gestalt**


---

`Gestalt(selector)`

Returns information about the Newton system; the type of information returned depends on the value of the *selector* parameter.

*selector*                      A constant that specifies the type of information that is returned on the system. The following values are



## Miscellaneous

**currently allowed:** `kGestalt_SystemInfo`,  
`kGestalt_Backlight`, `kGestaltArg_HasSoftContrast`, and  
`kGestaltArg_VolumeInfo`.

**return value** Depends on *selector*; see DISCUSSION.

## DISCUSSION

The return value of this function depends on the value of *selector*, as follows:

- If *selector* is `kGestalt_SystemInfo`, `Gestalt` returns a frame with the following slots:

## Slot Descriptions

<code>manufacturer</code>	An integer indicating the manufacturer of the Newton Device.
<code>machineType</code>	An integer indicating the hardware type this ROM was built for.
<code>ROMStage</code>	A decimal integer indicating the language (English, German, French) and the stage of the ROM (alpha, beta, final).
<code>ROMVersion</code>	A packed integer indicating the major and minor ROM version numbers. You can use the following function to convert this number into an array containing integers for the ROM major and minor version numbers:

```
func (ROMVersionInteger)
begin
    local minor := BAND(ROMVersionInteger, 0xFFFF);
    local major := BAND(ROMVersionInteger>>16, 0xFFFF);
    [ Floor(StringToNumber(BAND(major>>12, 0xF)
        & BAND(major>>8, 0xF)
        & BAND(major>>4, 0xF)
        & BAND(major, 0xF))),
      Floor(StringToNumber(BAND(minor>>12, 0xF)
        & BAND(minor>>8, 0xF)
        & BAND(minor>>4, 0xF)
```

## Miscellaneous

```

                                & BAND(minor, 0xF)))]
end

```

Here is another example of code to test if your Newton is running OS 2.x. The following expression evaluates to a non-nil value if the major version is 2:

```
BAND((Gestalt(kGestalt_SystemInfo).ROMVersion)>>16, 0xFFFF) = 0x0002
```

**IMPORTANT**

Do not assume that if the Newton is running version 2.0 or later that a particular feature exists. You still need to test the Newton to make sure the feature exists. ♦

**Note**

The `machineType`, `ROMStage` and `ROMVersion` slots provide internal configuration information and should not be relied on. ♦

<code>screenWidth</code>	<p>An integer representing the width of the screen in pixels. The width takes into account the current screen orientation.</p> <p>For example, on the MessagePad 120, because the screen width is 240 and the screen height is 320, in portrait orientation <code>Gestalt</code> returns a width of 240. If the screen is rotated, <code>Gestalt</code> returns a width of 320.</p>
<code>screenHeight</code>	An integer representing the height of the screen in pixels.
<code>screenResolutionX</code>	<p>An integer representing the number of horizontal pixels per inch. For screens with square pixels, <code>screenResolutionX</code> equals <code>screenResolutionY</code>. On the MessagePad 120, for example, both <code>screenResolutionX</code> and <code>screenResolutionY</code> equal 85.</p>
<code>screenResolutionY</code>	An integer representing the number of vertical pixels per inch.
<code>screenDepth</code>	The bit depth of the LCD screen. For the MessagePad 120, the LCD supports a monochrome screen depth of 1.

## Miscellaneous

	The eMate 300 and MessagePad 200 have 4 bit depth LCD screens.
<code>patchVersion</code>	Returns 0 on an unpatched Newton and nonzero on a patched Newton.
<code>ROMVersionString</code>	The user-visible string that identifies the version of the installed ROM and the installed patch, if any. The first part of the string is a “functionality level” indicating the OS version, such as 1.3, 2.0 or 2.1. The second part of the string is a six-digit number in parentheses that is an encoded representation of ROM and system update information.
<code>cpuType</code>	A symbol specifying the type of CPU, possible values are 'strongArm, 'arm710a, and 'arm610a.
<code>cpuSpeed</code>	A real indicating the speed of the CPU in megahertz.

- If *selector* is `kGestalt_Backlight`, `Gestalt` returns either `nil`, indicating the unit does not have backlight hardware, or a one element array. If an array is returned, the unique element contains either `nil` or a non-`nil` value, indicating whether backlight hardware is present.

The following code correctly tests if a unit has a backlight:

```
local result := Gestalt(kGestalt_Backlight);
if result and result[0] then
    // unit has backlighting
else
    // unit does not have backlighting
```

- If *selector* is `kGestaltArg_HasSoftContrast`, `Gestalt` returns either `nil`, or a 3-element array of the following form:  
`[hasSoftContrast, minContrast, maxContrast]`

**Array Element Descriptions**

<i>hasSoftContrast</i>	True or <code>nil</code> depending on whether there is a soft contrast control.
<i>minContrast</i>	Integer for the minimum contrast.
<i>maxContrast</i>	Integer for the maximum contrast.

## Miscellaneous

You can use the values returned by this selector to set the `LCDDContrast` user configuration variable.

- If *selector* is `kGestaltArg_VolumeInfo`, `Gestalt` returns either `nil`, or a 7-element array of the following form:  
`[hasInput, hasOutput, hardwareVolControl, headphoneJack, minAudibleDB, numDVLevels, devicesBitfield]`

**Array Element Descriptions**

<i>hasInput</i>	True or <code>nil</code> depending on whether the device can support sound input.
<i>hasOutput</i>	True or <code>nil</code> depending on whether the device can support sound output.
<i>hardwareVolControl</i>	True or <code>nil</code> depending on whether the device has a hardware volume control.
<i>headphoneJack</i>	True or <code>nil</code> depending on whether the device has a built-in headphone jack.
<i>minAudibleDB</i>	An integer, the minimal decibel level for output. The MessagePad 2000 is set to -31.9760.
<i>numDVLevels</i>	An integer, the number of levels between <i>minAudibleDB</i> and 0. The dB increment per level is <i>minAudibleDB</i> / <i>numDVLevels</i> . The MessagePad 2000 is set to 14.
<i>devicesBitfield</i>	A packed integer with information about the built-in sound devices. This integer contains the summation of the applicable device constants. Device constants are described in “Device Constants” (page 7-26) in Chapter 7, “Sound.” The two important ones are <code>kInternalSpeaker</code> and <code>kInternalMic</code> .  The following function returns <code>nil</code> or <code>non-nil</code> , indicating if the current device has an internal microphone (use <code>kInternalSpeaker</code> to check for an internal speaker):

```
HasMic := func()
begin
    local volInfo := Gestalt(kGestaltArg_VolumeInfo) ;

    return volInfo AND
```

## Miscellaneous

```

        (BAND(volInfo[6], kInternalMic) <> 0);
    end

```

**COMPATIBILITY**

The `kGestalt_Backlight` and `kGestaltArg_VolumeInfo` selectors are not supported on 2.0 devices.

**TimeFrameStr**


---

`TimeFrameStr(timeFrame, timeStrSpec)`

Returns a string representation of the time *timeFrame*, in the specified format.

<i>timeFrame</i>	A date frame as returned by the <code>Date</code> function.
<i>timeStrSpec</i>	A format specification returned by the <code>GetStringSpec</code> function, or one of the format specifications found in <code>ROM_dateTimeStrSpecs</code> .
return value	A string representation <i>timeFrame</i> .

**DISCUSSION**

This function is similar to the `TimeStr` function. `TimeStr` is passed in the time as an integer, in minutes. Thus, when a format spec is provided that requires seconds, `TimeStr` returns a string with 00 as the seconds value. `TimeFrameStr`, on the other hand, since it is passed the time as a date frame, can include seconds information.

## Summary

---

### Error Codes

---

There are two new NewtonScript Environment error codes:

-48034	Soup name too big
-48426	Unexpected rich string

### Data Structures

---

#### View Slot

---

```
hilitedData // true = view has data that can be cut or copied
```

#### Clipboard Data Frame

---

```
aClipboardDataFrame := {
  label: string, //string displayed by clipboard
  types: array, //array of types arrays
  data: array, //array of data arrays
  bounds: frame, //where data came from
  ...}
```

#### Extras Drawer Folder Symbols

---

```
nil
'_extensions
'_help
'_setup
'_soups
'_ButtonBar
```

#### Names Worksite Soup Entry

---

```
cityAlias // city information array
```

## Miscellaneous

**Newton Works Word Processor Soup Format**

---

```

aWorksWordProcessorSoupEntry := {
class: 'paper,
version: integer,
title: string,
timeStamp: integer,
realModTime: integer,
saveData: frame,
hiliteRange: frame,
margins: frame,
}

```

**User Configuration Variables**

---

```

LCDContrast
alarmVolumeDb
soundVolumeDb
buttonBarPositions
buttonBarControlsPositions
bellyButtonPositions
buttonBarIconSpacingH
buttonBarIconSpacingV
extrasIconSpacingH
extrasIconSpacingV
extraFont

```

**Dial-In Networks Access Frame**

---

```

aNetworkAccessFrame := {
mailNetwork: symbol, //network symbol
mailPhone: string, //phone number
baud: integer, //data rate supported
}

```

**Dial-In Networks Network Frame**

---

```

aNetworkFrame := {
title: string, //network name
id: symbol, //unique network id
GetAccessNumbers: func(worksiteFrame, cityFrame)..., //retrieves numbers
}

```

Miscellaneous

# Protos

---

## protoPasswordSlip

---

```
aPassWordSlip := {
  _proto: protoPasswordSlip,
  verifyPassword: symbolORtrueORnil, //should password be verified?
  CurrentPassword: func() ..., //gets curr password
  SetPassword: func(newPassword), //sets curr password
  MatchPassword: func(newPassword, currentPassword)...,//do these match?
  MatchedPassword: func() ... , //called if there was a match
  ...}
```

## protoBlindEntryLine

---

```
aBlindEntryLine := {
  _proto: protoBlindEntryLine,
  dummyChar: character, //char to echo
  realText: string, //the real text
  label: string, //entry line label
  UpdateText: func (newText), //updates text
  ...}
```

# Constants

---

## Screen Orientation Constants

---

Constant	Value
kPortrait	0
kLandscape	1
kPortraitFlip	2
kLandscapeFlip	3

## Serial Communication Tool Sound Option

---

kCMOPCMCIAModemSound	"msnd"
----------------------	--------



## Miscellaneous

## Functions and Methods

---

### Views

---

```
view:DragAndDrop(unit, dragBounds, pinBounds, copy, dragInfo)
    //starts the drag and drop process (2.0 also)
view:DragAndDropLtd(unit, dragBounds, limitBounds, copy, dragInfo)
    //starts the drag and drop process in limited area (platform file)
view:ViewAddDragInfoScript(dragInfo) //called if hilitedData is true
```

### Stationery

---

```
RegStationeryChange(regSymbol, functionBody)
    //regs callback for stationery change
UnRegStationeryChange(regSymbol) //unregs a stationery change callback
```

### Text

---

```
MakeFontMenu(font, families, sizes, styles) //makes a font menu
```

### Recognition

---

```
RecognizeTextInStyles(textFrame, defaultFontSpec)
    //recognizes ink in a frame
RecognizeInkWord(inkWord) //recognizes an ink word
```

### System Services

---

```
BatteryStatus(which) //returns info about a battery (2.0 also)
```

### Built-in Applications and System Data

---

```
extrasDrawer:GetPartEntryData(entry)
    // gets info about a part entry (platform file - 2.0 also)
calendar:SetEntryAlarm(mtgText, mtgStartDate, minutesOrDaysBefore)
    // sets an alarm for a meeting or event (2.0 also)
SetUserConfigEnMasse(changeSym, changeFrame)
    // sets multiple user configuration variables
RegUserConfigChange(callBackID, callBackFn)
    //registers a callback for changes in a user configuration var.
KillStdButtonBar(buttonBarParams)
    // closes (or restores) the button bar
```

## Miscellaneous

**buttonBar**:GetPartEntries() //returns part entries for parts in b. bar  
**buttonBar**:ReConfigure(**newSetup**) //reconfigures the button bar  
**buttonBar**:IconCapacity() // gets number of icons that fit in button bar

## Transports

---

**transport**:DeleteItem(**item**) //deletes item from In/Out box  
**transport**:DeleteRemoteItems() //deletes remote items  
**transport**:RefreshOwner() //refreshes the transport owner

## Dial-In Networks

---

RegDialinNetwork(**networkSym**, **networkFrame**) //regs new dialin network  
 UnRegDialinNetwork(**networkSym**) //unregs dialin network  
 GetLocAccessNums(**entry**, **which**) //returns array of access frames  
 GetAllDialinNetworks() //returns array of all dialin network frames  
 GetDialinNetwork(**networkSym**) //returns a dialin network frame

## Utility Functions

---

GetClipboard() //returns the contents of the clipboard  
 SetClipboard(**clipboardData**) //sets the contents of the clipboard  
 ROM\_GetSerialNumber() //gets a unit's unique serial number  
**partFrame**:ImportDisabled(**unitName**, **majorVersion**, **minorVersion**)  
     //called to clean up when unit is disabled (2.0 also)  
 LegalOrientations() //gets legal values for screen orientation  
 SetScreenOrientation(**orientation**) //sets sceen orientation  
 GetAppParams() //gets info about app area and button bar (2.0 also)  
 Gestalt(**selector**) //gets info about the system (2.0 also)  
 TimeFrameStr(**timeFrame**, **timeStrSpec**) //returns string with time