

# Misc ToolBox Calls

version 0.1 January 27, 1997

1.0 Introduction.....	1
2.0 Hardware Serial Number.....	1
3.0 Unit Import/Export.....	1
4.0 Screen Orientation.....	2
5.0 User Configuration .....	2
5.1 Volume and LCD APIs.....	3
6.0 GetAppParams .....	3
7.0 New Gestalt Selectors.....	4
8.0 Button Bar APIs.....	5
9.0 Part Cursor Additions .....	5
10.0 Ink Recognition functions.....	5
11.0 Extensions to DragAndDrop .....	6
12.0 Clipboard API.....	8
12.1 Clipboard Data Frame.....	8
12.2 Clipboard API.....	8
13.0 TimeFrameStr.....	9
14.0 New Font APIs.....	9
15.0 Password slip.....	10
15.1 blindEntryLine.....	11
16.0 Stationery Change API.....	12
17.0 Battery API.....	12
18.0 Grayscale API.....	12

## 1.0 Introduction

This documents lists a number of miscellaneous toolbox calls that were left out of the current revision of the Newton 2.1 OS documentation. Unless otherwise stated, all calls exist only in Newton OS 2.1 or better. If you intend to write code that will also run on Newton OS 2.0, you first check with `GlobalFnExists` before using the function.

## 2.0 Hardware Serial Number

One of the new features in Newton OS 2.1 is the ability to support a unique serial number chip. Both the MessagePad 2000 and eMate 300 will have these chips. Since each unit has a unique number it can be used for inventory tracking, copy protection, or just geek factor. Note that the number on the ROM is not the same as the serial number shown on the outside of the unit. The ROM unique ID is meant for programmers to use.

You access the serial number by calling a function that is provided as a magic pointer. The function is accessed by the constant `ROM_GetSerialNumber`. This constant is defined in the Newton OS 2.1 platform file. The function will return a binary object that is 8 bytes long. If there is no serial number chip, the function will return nil.

As an example of usage, you can get a human readable hex dump of the serial number by using the `StrHexDump` function. So an example call would be:

```
StrHexDump(call ROM_GetSerialNumber with (), 0)
#C41A2E5 "0000000001548423"
```

**Note:** Since your code is likely to run on non-OS 2.1 units, you should check to make sure the magic pointer is valid by wrapping your call in a try/onException block:

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

if sn then
    // ...
```

## 3.0 Unit Import/Export

There is one modification to the Unit Import/Export APIs. For more information on Unit Import/Export, see the MooUnit DTS Sample code.

```
ImportDisabled(unitName, majorVersion, minorVersion)
```

Previously, the return value of this script was unspecified. You can now optionally return the symbol `'ThrillMeChillMeFulfillMe'` which will cause the system to attempt to re-resolve the imports.

For example if version 2 of unit foo is disabled and your packages `ImportDisabled` script returns `'ThrillMeChillMeFulfillMe'` then the system will look for other versions of the objects in the unit foo (assuming they exist).

## 4.0 Screen Orientation





`LegalOrientations()`

Returns an array of the legal values for orientation of the screen on the Newton Device. Each value of the array is an integer. See `GetOrientation` for currently supported values.

**Note:** This function is supported in Newton OS 2.0. On the MessagePad 120 and 130 units, the only possible return values are 0 and 3.

`GetOrientation()`

Returns an integer that specifies the current orientation of the unit. There are currently 4 possible integer return values:

kPortrait 0	kLandscape 1	kPortraitFlip 2	kLandscapeFlip 3
	 NA	 NA	

**Note:** This function is supported in Newton OS 2.0. On the MessagePad 120 and 130 units, the only possible return values are 0 and 3.

`SetScreenOrientation(orientation)`

*orientation*            An integer specifying the new orientation (see `GetOrientation`)

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.

The function returns a non-nil value if the screen orientation was changed. A return value of nil means that the orientation was not changed.

## 5.0 User Configuration

`SetUserConfigEnMasse(changeSym, changeFrame)`

*changeSym*            symbol passed to functions registered for notification of user configuration changes

*changeFrame*        frame specifying which slots to set and new values (see below)

This function lets you set a multiple user configuration slots with a single call. The `changeFrame` argument specifies which slots to change and what values to use. Each slot in the frame corresponds to the user configuration value you wish to change. The value in the slot is used as the new value. As an example, to set both the system wide alarm volume and the LCD Contrast the call would be:

```
SetUserConfigEnMasse('alarmVolumeDb,  
    {alarmVolumeDb: newVolume,  
    LCDContrast: newContrast});
```

**Note:** To accommodate the this new call, the function object passed to `RegUserConfigChange` can optionally have a second argument. This will be the `changeFrame` passed to `SetUserConfigEnMasse`. This argument is only supported in systems that have `SetUserConfigEnMasse`, so check for the function before registering a callback with 2 arguments.

## 5.1 Volume and LCD APIs

The correct way to set the volume and LCD contrast is via `SetUserConfig` - there are change handlers in the ROM that will propagate the changes to the hardware. Volume is now set in dB (values ranging from 0 to -infinity, use `Gestalt` to find the current range). The old slots are maintained for backward compatibility. The new slots in user configuration are:

`LCDContrast`

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 `kGestaltArg_HasSoftContrast` `Gestalt` selector (see 7.0 New Gestalt Selectors).

`alarmVolumeDb`

Sets the system wide alarm volume in decibels. Use the `kGestaltArg_VolumeInfo` `Gestalt` selector to find the range of values for the volume (see 7.0 New Gestalt Selectors).

`SoundVolumeDb`

Sets the system wide volume in decibels. Use the `kGestaltArg_VolumeInfo` `Gestalt` selector to find the range of values for the volume (see 7.0 New Gestalt Selectors).

## 6.0 GetAppParams

`GetAppParams` now returns additional information. The new or modified slots are:

`appAreaGlobalTop`

The coordinate of the top of the application area in global coordinates.

`appAreaGlobalLeft`

The coordinate of the left of the application area in global coordinates.

buttonBarPosition

It is now possible for the value of this slot to be 'none'. This means there is no button bar. See the article references in section 8.0 for more information.

buttonBarBounds

If there is a button bar this slot contains the view bounds of the button bar relative to the application area. Note that this slot is only valid in OS 2.1 or better. See the article references in section 8.0 for more information.

## 7.0 New Gestalt Selectors

There are a few new Gestalt selectors for OS 2.1 functionality. They are defined in the Newton 2.1 Platform file.

kGestaltArg\_HasSoftContrast

Returns an array of 3 elements:

```
[<has soft contrast>, <min contrast>, <max contrast>]
```

The first item is true if there is a soft contrast control. The other two elements give the integer values for the minimum and maximum contrast.

You can use the values returned by this selector to set the LCD contrast using the user configuration information in section 5.1 above.

Here is a sample call using an inspector connected to a MessagePad 2000:

```
Gestalt(kGestaltArg_HasSoftContrast)
#C63C125 [TRUE, -16, 16]
```

kGestaltArg\_VolumeInfo

Returns an array of 7 elements:

```
[<has-input>, <has-output>, <hardware-vol-ctl>,
<headphone-jack>, <min-audible-db>,
<num-db-levels>, <devices-bitfield>]
```

This selector is used to find information about the sound capabilities of a Newton device. It can be used to determine the range of output sounds in decibels (for use with setting the user configuration volumes; see 5.1 above).

has-input	true if the device can support sound input
has-output	true if the device can support sound output
hardware-vol-ctl	true if the device has a hardware volume control
headphone-jack	true if the device has a built-in headphone jack
min-audible-db	minimal output sound value in decibels (see Note below)
num-db-levels	number of levels between min-audible-db and 0 (see Note below)

devices-bitfield    information about built-in sound devices

The devices-bitfield contains information about the built-in sound devices. The constants used are the same as those used by the new Sound API. The two important ones are kInternalSpeaker and kInternalMic.

To find out if a unit has an internal microphone , you can use the following code:

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

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

This will return true if the unit has a built-in microphone. By substituting kInternalSpeaker for kInternalMic you can do the same thing for detecting a built-in speaker.

Here is a sample call using an inspector connected to a MessagePad 2000:

```
Gestalt(kGestaltArg_VolumeInfo)
#C63CDB1 [TRUE, TRUE, NIL, NIL, -31.9760, 14, 29]
```

**Note:** the volume is set in decibels. This Gestalt selector provides the minimal decibel level for output (min-audible-db) and the number of increments between the minimal and maximal dB level. Since the maximal output in dB is 0, you can find the dB increment per level from min-audible-db / num-db-levels.

## 8.0 Button Bar APIs

See the upcoming Newton Technology Journal article entitled "Behind Bars" by Mike Engber. This will appear in volume 3, number 2 (i.e., the next NTJ).

## 9.0 Part Cursor Additions

Passing the symbol '\_all for labels does not include the button bar icons. To find those you need to use the symbol '\_ButtonBar.

The frame returned by GetPartEntryData may have an additional slot in OS 2.1:

iconPro    a frame used for grayscale icons. The frame has 2 slots that give the hilited and unhilited state of the icon.

## 10.0 Ink Recognition functions

RecognizeTextInStyles(*textFrame*, *defaultFontSpec*)

*textFrame*            frame with 'text and 'styles slots

*defaultFontSpec*    (integer or frame) font spec to use for translated ink

If there are no ink words, returns the original frame. Otherwise returns a new frame with 'text and 'styles slots, containing translated versions of all the ink words. The first translation option is used for each ink word found.

RecognizeInkWord(*inkWord*)

*inkWord*                      ink word data from a rich string or from a style array

Returns an array of translation options, where each entry in the array is a frame with a translated string in the slot 'word. Returns nil if no translation options were found.

## 11.0 Extensions to DragAndDrop

There is a new view method in Newton OS 2.1 that will be covered by a platform file function for the 2.0 OS (as of this writing, the function is not in the platform file).

view:DragAndDropLtd(*unit*, *dragBounds*, *limitBounds*, *copy*,  
*dragInfo*)

*unit*                              same as DragAndDrop

*dragBounds*                      same as DragAndDrop

*limitBounds*                      see below

*copy*                                same as DragAndDrop

*dragInfo*                        same as DragAndDrop

return value                      same as DragAndDrop

DragAndDropLtd is similar to DragAndDrop except for the limitBounds argument. The limitBounds argument can either be the actual limit bounds (see below) or a frame containing a limitBounds and/or a pinBounds slot.

*limit bounds*

bounds frame

A rectangle (in global coordinates) to constrain the drag

nil (or omitted from pin/limit frame)

The limit bounds will be the app area

'none

The limit bounds will be the whole screen.

*pin bounds*

bounds frame

A rectangle specifying the bounds to use when constraining the the object being dragged within the limit rect. Normally, this would be the same as the bounds of the object being dragged (i.e. the drag bounds), but if the object is large (relative to the limit rect) it may be desirable to specify a smaller rectangle or the object may appear not to move "far enough."

nil (or omitted from pin/limit frame)

The pin bounds will the drag bounds

'none

The pin bounds will be an empty rectangle at the place the pen went down. I.e. the object will move until the tip of the pen reaches the limit bounds.

**Note:** the limitBounds argument to DragAndDrop is incorrectly documented. It is actually a pin bounds and the legal values for it are a bounds frame or nil. DragAndDrop always uses the app area for limit bounds.

## DragAndDrop

The fourth argument to :DragAndDrop() is an array of "DragInfo" frames. This is not new. However, there is a new slot you can add to this frame, 'minDragDistance' which takes an integer value. This is the minimum distance in pixels that the user must drag the object before it really moves. The default value is 4.

DragAndDrop has been extended to work with the cut and copy global command keys. Before, they completely ignored "non-editView" views. There is now a new script, viewAddDragInfoScript() and a new slot, hilitedData.

## hilitedData

If a view has this slot defined and it is true, the cut and copy global command keys will be able to act on the view. That is, a true value informs the system that the non-edit view has something that can be cut and/or copied.

When the cut or copy key is used, the view will be sent a viewAddDragInfoScript. If the command key is a cut, you will also be sent a viewDropRemoveScript.

## viewAddDragInfoScript(dragInfo)

*dragInfo* the same as the parameter passed to DragAndDrop (see NPG)

Your view should determine if there is something to be "dragged", i.e., cut or copied. If so, create the drag info frame for that item and add it to the dragInfo array passed to the method. Return true if there is something to drag, nil otherwise.

As an example

```
myView.viewAddDragInfoScript := func(dragInfo)
begin
    // check and see if there is a selected item
    if currentSelection then
```



```

begin
    // yep, create an info frame using my own method
    local myInfoFrame := :GetDragInfoForSelection();

    // add it to dragInfo
    AddArraySlot(dragInfo, myInfoFrame) ;
    // and return true so the cut/copy happens
    return true ;
end ;
else begin
    // there is nothing selected
    // return NIL so the cut/copy does not happen
    return nil ;
end ;
end ;

```

## 12.0 Clipboard API

Newton 2.1 OS provides an API to get and set the contents of the global clipboard. The information on the clipboard is basically the same used by the Drag and Drop system, so you should understand that first.

### 12.1 Clipboard Data Frame

The information on the clipboard is represented in a frame with the following slots:

**label**

The text displayed by the clipboard item.

**types**

Array of types for the clipboard item (as per Drag and Drop)

**data**

Array of data. There is one item per type in the `types` array (as per Drag and Drop)

**bounds**

The rectangle of where the data came from in global coordinates.

**bits**

Ignore this slot. For `SetClipboard` this slot **must** be NIL.

### 12.2 Clipboard API

`GetClipboard()`

Returns a frame representing the current item on the clipboard, or NIL if there is nothing on the clipboard. The frame is a Clipboard Data Frame (see 12.1 above)

`SetClipboard(clipboardData)`

*clipboardData*      a Clipboard Data Frame (see 12.1 above)

This will set the contents of the clipboard to the data that is passed in. The text displayed by the clipboard item will be based on the label passed in. Because of this, you should make the label short.

Passing NIL to `SetClipboard` will clear the contents of the current clipboard. You can use this to perform a paste (i.e., use `GetClipboard` to get the contents, then use `SetClipboard` with NIL to clear the clipboard).

## 13.0 TimeFrameStr

`TimeFrameStr(timeFrame, timeStrSpec)`

*timeFrame*      a date frame like one returned by the `Date` function

*timeStrSpec*      a `timeStrSpec` as per the `TimeStr` function

This function is almost identical to `TimeStr`, except that the returned string can have a valid seconds value. See the documentation to `TimeStr` in the Newton Programmers Guide for more information.

## 14.0 New Font APIs

As of Newton OS 2.1, the `Fonts` global variable is no longer supported. If you need to find out the installed fonts on a Newton Device, use the new `GetAllFonts` function. If you need to create a popup with available fonts, use the `MakeFontMenu` call.

`GetAllFonts()`

Returns an array of all the installed user fonts. The system font is not included in this list.

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

*font*      NIL or a font specification as either a frame or a packed integer that represents the default font. The returned font menu will check the items that correspond to the selected font family, size and style. Passing NIL results in no items being checked.

*families*      NIL, 'all, 'none, or an array of font families. Controls which fonts are returned. NIL uses the all user fonts in the system (recommended). 'all uses every font (including system font). 'none means don't include family choices in the returned menu. An Array specifies the list of font families to return for the menu.

*sizes*      NIL, 'none, or an array of numbers. NIL uses the sizes of the font argument. 'none means don't include size choices in the menu. An array specifies the list of sizes to return for the menu.

*styles*      NIL, 'none, or an integer. NIL uses the default styles in the system. 'none means don't include style choices in the returned menu. An integer is a bitField that specifies which styles should be returned. Use the `Font`

Face Constants (see NPG) for this bit field. For example, to display just bold and italic use `kFaceBold + kFaceItalic`.

`MakeFontMenu` returns an array of menu items for use by `PopupMenu` or `protoPopUpButton`. The array items can display all system choices for a font family, size and style. Calling `MakeFontMenu(nil, nil, nil, nil)` will return an array of all user fonts, families(plain, bold, italic, etc.) and sizes. None of the items will be checked.

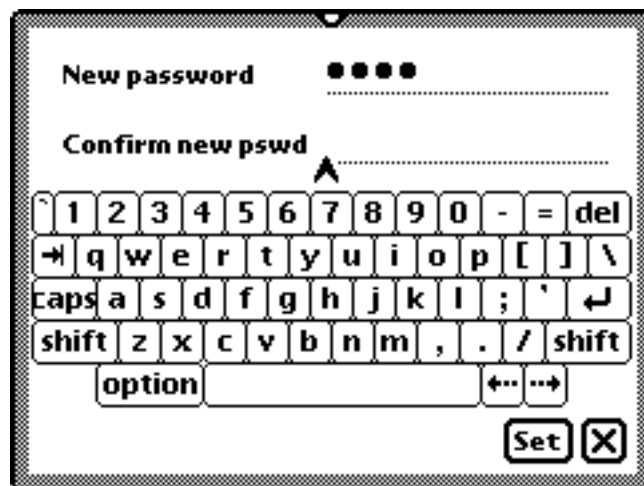
At this time, the styles argument ignores `kFaceSuperscript` and `kFaceSubscript`.

Here is a sample call using an inspector connected to a MessagePad 2000:

```
MakeFontMenu(nil, nil, nil, nil)
#C636259 [{item: "Casual", family: handwriting},
          {item: "Fancy", family: newYork},
          {item: "Simple", family: geneva},
          pickseparator,
          {item: "9 point", size: 9},
          {item: "10", size: 10},
          {item: "12", size: 12},
          {item: "18", size: 18},
          pickseparator,
          {item: "Plain", face: 0},
          {item: "Bold", face: 1},
          {item: "Italic", face: 2},
          {item: "Underline", face: 4},
          {item: "Outline", face: 8}]
```

## 15.0 Password slip

As of Newton OS 2.1 the system provides a proto that 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. If there is no attached keyboard, the slip has an embedded keyboard to allow typing.



To bring up the passwordslip, call `BuildContext` and `Open` on a template with the following slots defined:

`_proto`

Set to the value `protoPasswordSlip` (defined in the platform file)

`CurrentPassword( )`

Return the current password or nil if there is no current password.

`SetPassword(newPassword)`

*newPassword*      the new password to check for as a string

Sets the new password that will be checked for. Note that the password is a string in plain text, so for maximum security it should be encoded before being stored.

`MatchPassword(newPassword, currentPassword)`

*newPassword*      password entered by user

*currentPassword*   current password as returned by `CurrentPassword`

Return true if the two match, nil if not.

This method is called to verify that the correct password has been entered.

`MatchedPassword( )`

Called if a valid password was entered. You must call the inherited method to correctly close the password slip.

`verifyPassword`

a symbol whose value determines if the password slip is used to just ask for a password or if it is used to change a password.

A value of `verifyOnly` specifies a password slip that will query a user for a password. A value of true or NIL specifies a password slip that will prompt the user to change their password. True means the user must verify the old password (this is the default). NIL means the user does not need to verify the old password.

## 15.1 blindEntryLine

The `blindEntryLine` proto is used in the `protoPasswordSlip`. `BlindEntryLine` is useful when the user's typing should not be echoed on the line. Gestures are disabled on `blindEntryLines`, since the entered text must be kept in sync with the user's keydowns. To create a `blindEntryLine`, create a template with the following slots:

`_proto`

Set to the value `protoBlindEntryLine` (defined in the platform file) (as of this writing, the current 2.1 platform file has `blindEntryLine`, a bug has been filed)

`dummyChar`

This slot is optional. If present it is a character containing the text to display instead of the real text. By default, the bullet character is used.

`UpdateText (newText)`

*newText*                a string that is the new value for the `blindEntryLine`

This method will set the value of the `realText` slot (see below) to the value in `newText` and correctly update the string displayed to the user.

The proto contains an additional slot which is set to the string typed in by the user. You should not set this slot directly, use the `UpdateText` method instead.

`realText`

contains the string that the user has typed. You should use this slot for looking up the value of the text (instead of looking in the text slot).

Do not modify this slot directly. Use the `UpdateText` method.

## 16.0 Stationery Change API

`RegStationeryChange (regSymbol, functionBody)`

*regSymbol*                unique symbol that includes your signature

*functionBody*            a function body that is called when stationery changes

This function lets you register a function body that will be called when stationery is installed or removed. The function body takes four arguments:

`message`    currently 'install or 'remove

`defType`    currently 'dataDef or 'viewDef for the type of stationery that has been installed or removed.

`symbol1`    The dataDef symbol of the installed or removed stationery.

`symbol2`    If `defType` is 'dataDef then this is undefined. If `defType` is 'viewDef then this is the viewDef symbol of the installed or removed stationery.

`UnRegStationeryChange (regSymbol)`

*regSymbol*                symbol used to register for change notification

Unregisters a function body previously registered using `RegStationeryChange`.

## 17.0 Battery API

The possible slots in the frame returned by `BatteryStatus` have been extended. More information will follow later.

## 18.0 Grayscale API

The function `GrayShrink` referred to in the 2.1 documentation was innadvertantly left out of the documentation. It is documented here:

```
view:GrayShrink(bitmap, style)
```

*bitmap*                    A 1-bit pix family or bitmap graphic shape.

*style*                    A style frame, as used by `DrawShape`. This frame should contain a `transform` slot representing a reduction in size, either horizontally, vertically, or both. The `transform` slot should contain two bounds frames, i.e. not two integers. You may pass `nil` for the source bounds frame, in which case, `bitmap`'s bounds are used. You may also pass `nil` for the destination bounds, in which case the `viewBounds` slot is used.

Anti-aliases a 1-bit bitmap and renders it on the screen.

If *bitmap* is not 1-bit, or if *style* does not have a `transform` slot representing a reduction in size, `bitmap` is still rendered on the screen, but not anti-aliased.