

# NOTICES

Copyright © 1993-1996 Jason Harper.  
All rights reserved.

ViewFrame is a trademark of Jason Harper. Apple and Newton are trademarks of Apple Computer, Inc., registered in the United States and other countries. MessagePad, NewtonScript, and Newton Toolkit are trademarks of Apple Computer, Inc.

ViewFrame is capable of being used in ways which might constitute infringement of licensing agreements or copyrights belonging to Apple or other parties. ViewFrame does not enforce any restrictions on the objects that can be viewed. It is entirely the responsibility of the user to determine the legality of any particular use of ViewFrame, and to face any consequences of such use.

Please note that anything you might learn about the internal operation of Newton devices by using ViewFrame may not be applicable to programs that you write – system software operates under different rules than user programs. In particular, the system software knows what hardware it is running on, and can safely assume that the hardware will remain constant (since any future hardware will come with system software customized for it). Your programs, on the other hand, may some day be run on devices with different hardware, or under system software with a different set of features.

As an example, you would probably find that Newton system software through version 1.3 makes the assumption that there can be only two stores, internal and external. This is perfectly fine, for the system software, since this is a physical limitation of the hardware it is designed to support. However, if you made the same assumption in your program, it would fail on any future Newton devices with a different hardware configuration, such as having two PCMCIA slots.

Writing programs with no consideration for future compatibility may be easier in the short term, but will cause you troubles down the line as Newton technology evolves.

## WHAT'S NEW IN VERSION 1.2 4

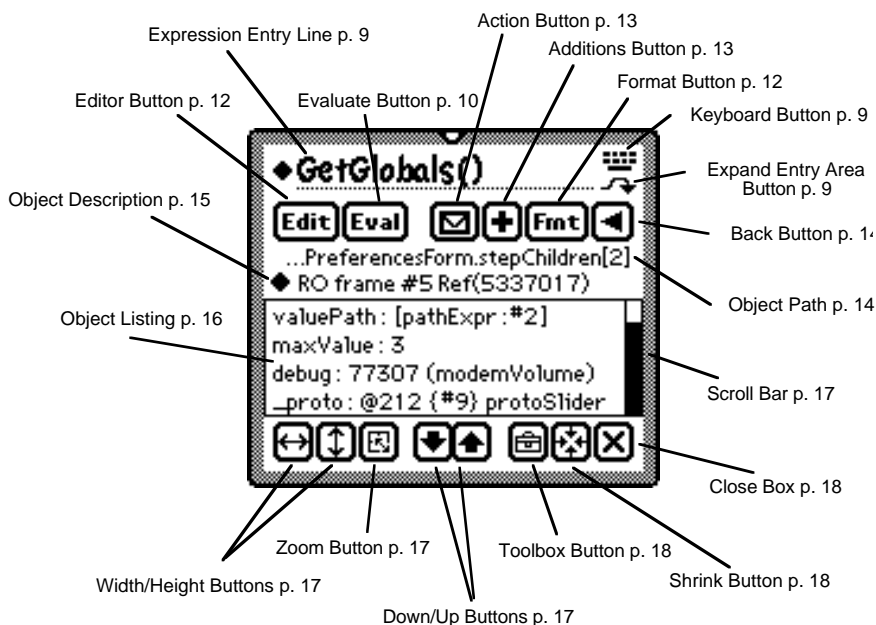
## GETTING STARTED 5

*Items Required to Use ViewFrame • Recommended Items  
Installation • Quick Start*

## BASIC CONCEPTS 7

*Current Object • Object Path • Magic Pointers*

## USING VIEWFRAME 8



## CONTENTS

<b>THE VIEWFINDER</b>	<b>19</b>
<i>ViewFinder and Palette Usage • Drag &amp; Drop</i>	
<b>VIEWFRAME ACCESSORIES</b>	<b>23</b>
<i>ViewFrame Editor • Programmer's Keyboard</i>	
<b>INTERPRETING RESULTS</b>	<b>31</b>
<i>Object Descriptions • Object Listings • Generic Frame/Array Listings</i>	
<i>Location Listings • Function Listings</i>	
<b>TIPS &amp; TECHNIQUES</b>	<b>40</b>
<i>Finding an Application • What Am I Looking At?</i>	
<i>Browsing Soups • UNABLE TO BUILD LIST</i>	
<b>ADVANCED USES</b>	<b>44</b>
<i>Directly Using ViewFrame • Opening the Programmer's Keyboard</i>	
<i>Customizing ViewFrame • Writing Add-ons for ViewFrame</i>	
<b>NEWTONSCRIPT SECRETS</b>	<b>48</b>
<i>References • Trivia</i>	
<b>TECHNICAL SUPPORT</b>	<b>51</b>
<b>SOFTWARE LICENSE</b>	<b>52</b>


## WHAT'S NEW IN VERSION 1.2



The main new feature in ViewFrame 1.2 is support for the new Newton 2.0 operating system. Support for Newton 1.x systems remains, but is unlikely to increase beyond its current level in future versions of ViewFrame. Newton 2.0 presents many challenges and opportunities for a debugging tool. Some examples are:

- New types of objects, like virtual binary objects, entry aliases, rich strings, to recognize and display;
- New formats for some existing object types like functions, weak arrays;
- New possibilities for acquiring and interpreting objects such as Drag & Drop and ObjectPkgRef(); and
- Environmental changes, like screen rotation and a wider border on floaters, that affect all applications.

In general, support for Newton 2.0 features that are modifications or extensions of previous features is contained directly in ViewFrame and existing Addition packages, and used conditionally on Newton 2.0 systems. Support for brand new Newton 2.0 features is contained in the VF+Dante Addition package, which runs only on Newton 2.0 systems.

The most noticeable new feature for Newton 1.x systems is the ViewFinder, a tool for visually locating open views that you can examine using ViewFrame. Other new features that are available on all Newton systems include a way of determining the point in a function that corresponds to a certain PC value (as shown in a stack trace), and some extensions to the ViewFrame Additions mechanism. Some additional changes are purely for convenience: the expression entry area can now be expanded to multiple lines, the scroll bar is slightly wider to make it easier to hit with the pen, and the NTK Toolkit App can be opened directly from ViewFrame.

ViewFrame features which require Newton 2.0 to work are indicated with the symbol . On Newton 1.x systems, these features do nothing, or simply don't appear at all. Note that these features may be useful even when working on a program that isn't Newton 2.0-specific. For example, the ability to determine exactly which package contains a given object works only on a Newton 2.0 system, but it works even if the package in question is intended for Newton 1.x use.

All features flagged with  are new, since this is the first version of ViewFrame to support Newton 2.0. Other new features in this version are flagged with  wherever they are mentioned in this manual.

## GETTING STARTED

Read the “Read Me.txt” file on the ViewFrame disk, if present. It contains corrections to this manual, and updated information for program versions more recent than the ones covered here (ViewFrame 1.2, ViewFrame Editor 1.1, Programmer’s Keyboard 1.1).

### ITEMS REQUIRED TO USE VIEWFRAME

You need the following items to use ViewFrame:

- A Newton device. ViewFrame was developed and tested on various MessagePad models, using system versions up through 2.0. It should be at least partially functional on any future Newton model. ViewFrame has to use some undocumented features in order to do what it does – full future compatibility cannot be guaranteed.
- A basic knowledge of the NewtonScript language. Sources of documentation are the Newton Toolkit (NTK), the Newton programming books from Academic Press, and *PDA Developers* magazine, published by Creative Digital, Inc.
- Some method of downloading packages to your Newton. The NTK, Newton Connection Kit, and a few other utilities are capable of doing this.

### RECOMMENDED ITEMS

We also recommend that you have the following items when using ViewFrame:

- A Newton storage card. A full ViewFrame installation takes over 250K, which may leave little room for other programs or data.
- Some way to back up your Newton data in the event your data gets corrupted (this can occur by careless use of ViewFrame). A storage card or the Newton Connection Kit is adequate.
- The Inspector, for Newton debugging from a host computer. This is part of the NTK, with no other equivalents currently available.
- A printout of the NTK Definitions file appropriate for your Newton model. This is often useful for interpreting magic pointer references that ViewFrame displays.

## INSTALLATION

ViewFrame consists of several separate packages:

- ViewFrame, for examining and modifying NewtonScript objects.
- ViewFrame Editor, for writing simple programs directly on a Newton device. The Editor requires ViewFrame for displaying program results other than simple types such as integers.
- VF Ed SoupMaker, which installs a few sample programs for the Editor and then deletes itself.
- Programmer's Keyboard, for easy on-screen entry of NewtonScript expressions in ViewFrame or the Editor. The normal on-screen keyboard is used if this package isn't installed.
- Various ViewFrame Addition packages, containing add-on object viewers and commands for ViewFrame. Since the contents of the Addition packages are expected to change fairly often, full documentation on them is supplied only on disk in the file "Additions.doc".

You can install whatever subset of these packages you need, and you can freely mix them between internal and card storage. On Newton 1.x systems with a nearly full Extras Drawer, you should install ViewFrame first. It doesn't matter if the Editor or Keyboard are inaccessible from the Extras Drawer – the Editor can be accessed directly from ViewFrame, and the Keyboard can be accessed from either ViewFrame or the Editor. ViewFrame Additions are auto part packages which normally don't show up in the Extras Drawer (on Newton 2.0 systems, switch to the Extensions folder to see them).

## QUICK START

- Open your Extras Drawer and select ViewFrame, just as you would open any other installed software.
- Enter a NewtonScript expression on the top line by tapping the diamond to the left and choosing from the pop-up list, then tap Eval. `GetRoot()` and `GetGlobals()` are good choices to start browsing.
- Look at slots in the current object by tapping on them. Return to the previous object by tapping on the left-pointing arrow.
- Scroll through long listings by using the arrows at the bottom of the ViewFrame window, *not* the arrows at the bottom of the Newton screen.

## BASIC CONCEPTS

### CURRENT OBJECT

The current object is the NewtonScript object that ViewFrame is working with presently. Details of the current object are displayed in the box that fills most of ViewFrame's window, and a brief description is displayed just above this box.

### OBJECT PATH

The object path is the list of objects that were viewed in the process of getting to the current object. The first object in the path is always generated by an expression entered into ViewFrame (or by another program telling ViewFrame to view a particular object). Subsequent objects are added to the path by expressions, or by selecting a slot in the current object by tapping on it. A list of the objects in the path appears above the description of the current object. If the list is too wide to fit in the ViewFrame window, objects at the left end of the list are omitted, since objects at the right end (the ones most recently viewed) are usually more important.

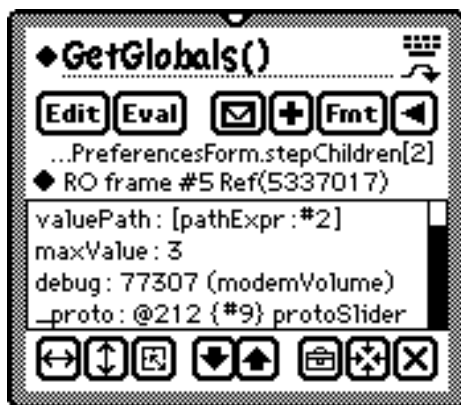
### MAGIC POINTERS

Magic pointers are the mechanism by which a program can reference built-in system objects (such as proto templates and sounds), without having to know the address at which the objects are located (which varies between Newton models and even between different ROM revisions of a single model). Magic pointers are constants, specifying a position within a table of system objects that is contained in each Newton ROM. They can refer to any NewtonScript memory objects (frames, arrays, and binary objects), but not immediate values. The syntax is an "@" (at sign) followed by an integer. Newton 1.x systems define magic pointers in the range @0 through @369. In Newton 2.0, there are over 800 defined magic pointers.

In Newton programming using the NTK, you generally use symbolic constants (which are listed in the NTK Definitions file) rather than write magic pointers directly. However, these constants do not exist in the Newton ROM, so ViewFrame can only display magic pointers (and allow you to enter them in expressions) in the @*nnn* form. Some objects referenced by magic pointers (especially proto templates) may contain a debug slot giving the object's name, or an encoded integer that can be turned into the object's name with Apple's DebugHashToName package. To interpret other magic pointer references, look them up in the NTK Definitions file.

Newton 2.0 extends the magic pointer concept to handle references between packages, not just references from packages to ROM. Inter-package magic pointers are implemented in a significantly different way – they are replaced by an actual reference to the destination object at the time the package is installed, rather than being looked up in a table each time they are used. For this reason, ViewFrame normally does not give any special indication of an inter-package magic pointer: there simply isn't anything to distinguish it from any normal reference.

You may occasionally see a magic pointer in the form @nnnn, where the number is greater than 8000. This indicates an inter-package reference that is unresolved (the package actually containing the object is not currently installed), or a reference to certain portions of the system software that are implemented as packages contained in ROM.



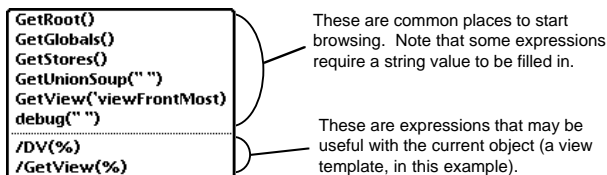


# USING VIEWFRAME

## EXPRESSION ENTRY AREA

### ◆GetGlobals()

Write or type in a NewtonScript expression here to specify an object to browse, or to perform some operation on the current object. Tapping the diamond pops up a list that contains commonly used starting expressions, and possibly some expressions relevant to the current object. The object types that ViewFrame has specific expressions for include view templates, views, stores, soups, and cursors.



If the current expression is a symbol which also exists in the root view (a common instance of this is an application symbol), the expression `/GetRoot().%` appears, which locates the associated application or other object in the root view.

## KEYBOARD BUTTON



Tapping the Keyboard button displays the Programmer's Keyboard (if it is installed) or the standard Newton keyboard. You may need to tap in the expression entry area to make the insertion point appear there. Text input in ViewFrame is most commonly done using a keyboard, since most NewtonScript expressions include words not in the Newton dictionary, or punctuation marks that Newton handwriting recognition doesn't support. However, Newton 2.0 owners may find that the new printed handwriting recognizer is adequate for expression entry.

If you have the Programmer's Keyboard installed, but would rather use the standard keyboard, you can bring it up by double-tapping in the expression entry area.

## EXPAND ENTRY AREA BUTTON



Tapping this button gives you more room for entering an expression. For each tap, the expression entry area grows by two lines, and the object display area shrinks by three lines, which preserves the overall height of the ViewFrame window. If the object display area would become less than three lines high, the height of the ViewFrame window is increased instead. If the window already fills the entire height of the screen, this button does nothing.

## SHRINK ENTRY AREA BUTTON



This button only appears if the Expand Entry Area button has been used to enlarge the expression entry area beyond its default size of one line. Each tap of this button undoes the effect of one tap of the Expand button: the expression entry area shrinks by two lines, and the object display area grows by three lines. Note that this button does not undo any enlargement of the ViewFrame window caused by the Expand button.



## EVALUATE BUTTON

Tapping Eval causes ViewFrame to compile and execute the current expression in the expression entry area. The effect on the previously displayed object depends on the first character of the expression, as follows:

- If the expression starts with anything other than one of the special characters described below, it is evaluated as-is. The result becomes the only item in the object path. The previous object path is discarded.
- If the expression starts with a “.”, “[”, or “:” (period, open square bracket, or colon), it is evaluated in the context of the current object, which must be a frame or array for this to make any sense. The result is appended to the object path, unless it’s the same as the current object. You can return to the previously displayed object by tapping the Back button. Some examples of this feature are:


<code>.viewChildren</code>	Look at a slot in the current object, without having to locate it in the listing of the object.
<code>[42]</code>	Same, for arrays.
<code>.text := "Test"</code>	Create or change a slot.
<code>[3] := nil</code>	Change an array element.
<code>:Hilite(true)</code>	Send a message to the current object.

- If the expression starts with a “/” (slash), things start getting interesting. The slash is removed, and each occurrence of “%” (percent sign) in the expression is replaced by a reference to the current object. The reference is enclosed in parentheses, which is occasionally significant. The result of evaluating this is appended to the object path (unless it’s the same as the current object, in which case the current display is simply updated).

The most common use of this feature is to call a global function, passing it the current object as a parameter. For example, if the current object is a string, you can enter the expression `"/StrLen(%)"` to find its length, then tap the Back button to return to the string object. However, you can put a slash in front of any valid NewtonScript expression. It doesn't have to contain a percent sign – evaluating it appends to the object path rather than replacing the path. This is very useful for temporarily looking at some other area of the Newton object system, without losing your place in the current object.

As another example, if you need to look at something in the root view, enter the expression `"/GetRoot()"` (probably by selecting `"GetRoot()"` from the pop-up expression list, then using the keyboard to insert a slash in front of it), then tap Eval to start browsing from the root view. When you're done with that, you can just back up to your previous object, which remained in the object path all along.

*WARNING: If you overuse this path continuation feature you eventually run out of memory. Even if all of the objects you look at are in ROM, memory is still consumed since ViewFrame keeps a copy in RAM of the expression that references each object.*

-  If the expression starts with a `"="` (equals sign) and the current object is a frame, a filtered version of the current object is appended to the object path. Only slots with names that contain the specified text are present in the new frame. For example, if you are looking at a huge view template, you can enter the expression `"=view"` to limit the display to the standard view attributes and methods such as `viewFlags`, `viewSetupFormScript`, and so on. The new frame consists of all slots that contain `"view"` in their names (not necessarily at the beginning).

You may have noticed that there is a possible ambiguity with expressions starting with a square bracket. An expression like `"[1]"` can be interpreted either as a NewtonScript array-construction expression, or as a reference to an element in the current object (assuming that it's an array). ViewFrame currently makes no effort to guess the correct interpretation – it always treats expressions starting with a square bracket as an array element reference. If you want to have such an expression treated as an array constructor, you must either enclose it in parentheses or precede it with a slash.

## EDIT BUTTON

Tapping the Edit button closes ViewFrame and brings up the ViewFrame Editor, or displays an error message if the Editor isn't installed. You can use the Editor to enter expressions larger than the entry area, but it doesn't support any of the special starting characters for expressions described on the preceding page. See page 23 for complete details on using the Editor.

## FORMAT BUTTON

Tapping the Format button pops up a list of formatting options that can be applied the current object display. The options are divided into two groups, one controlling overall format of the display, and the other controlling sorting of the displayed items. The first group contains three formats: Normal, Alternate, and Location. Normal format is the default for all objects when initially displayed – it uses the most appropriate viewer for each type of object. Alternate format has the following effects on the display:

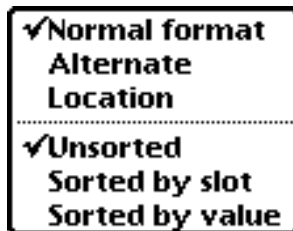
- Any objects which have specific viewers (icons, sounds, functions, and so on) are shown as the underlying frame or binary object that defines them.
- Frames are displayed deeply – all slots reachable from the object via prototype inheritance are shown. Slots not contained directly in the current object are indented according to the number of \_proto links that have to be followed to reach them.
- Binary objects such as strings and reals are displayed in hexadecimal, even if they have a direct printable representation.

**ONLY** The Location format uses new Newton 2.0 functionality to show you, on a slot-by-slot basis, exactly where an object is located: in RAM, in ROM, or contained in a particular package. It is useful for tracking down the “Grip Of Death” problem which occurs when a reference to a removed package is left in memory.

Reselecting the current format selection has the effect of rebuilding the display of the current object, showing any changes that have occurred to the object since the current display was created.

The sorting options in the second group can be selected independently of the format option, although not all combinations work well:


- Unsorted displays the contents of the object in their natural order.



- Sorted by slot arranges the listing so that the slot names are in alphabetical order. This only makes sense for frames (other object types simply get scrambled), and doesn't work very well for frames viewed deeply (by selecting the Alternate format). This is useful for locating a particular slot name in a large frame.
- Sorted by value rearranges the listing so that the slot values are in alphabetical order. This is only meaningful for frames and arrays. Note that this operates on the text representations of the slot values, since there is no comparison function that works on all object types directly. The ordering of slots after sorting isn't very meaningful, however all slots of similar type (such as frames, or functions) are grouped together afterwards.

## ADDITIONS BUTTON

Tapping this button pops up a list of commands contained in currently installed Addition packages. Only the commands relevant to the current object, or commands that don't require a current object, are shown at any given time. There is a dividing line between the commands of each additions package.

 All commands with names that start with "About" are collected into a separate list, accessible via a single "About..." item in the main Additions list. Generally, each package has an About command, which includes a copyright notice and a list of all the functions included in the package.

Commands may send information to the Inspector, display information in a separate view, add to or replace the current object path, or almost anything else. Additions that show information in the normal object display area usually work automatically and don't require any command to invoke them. See the Additions documentation on your disk for details on all the add-on commands supplied with ViewFrame.



## ACTION BUTTON

Tapping this brings up a list of options for producing external output from the current object. (The options may be slightly different under Newton 2.0.) You are probably familiar with its general use from other Newton applications that have an Action button.



### PRINT/FAX LISTING

These commands are for producing hardcopy of an object listing. You may choose either a one or two-column format. Two columns are generally good for long frame or array listings, since individual items are seldom wider than half the page. One column is often better for function listings, since they tend to contain long lines.

*WARNING: Printing or faxing hexadecimal binary objects can be very slow, due to the large amount of data generated. It may be impossible to fax a binary object in two column format, since the receiving fax machine may time out before the first band of data is sent. This is mainly a problem with older Newton devices, with system software prior to 1.3.*

### >/INSPECTOR

This choice sends the text of the current object listing to the NTK Inspector. Any graphics in the listing are omitted. This does nothing if your Newton device isn't currently connected to the Inspector. This is useful for looking at a listing that is too large for the Newton screen, or for comparing multiple listings. The format of the listing may not be exactly the same as on the screen. For example, hexadecimal listings show 16 bytes per line in the Inspector, instead of eight bytes per line on the screen.

### INSPECTOR PRINT()

This choice displays the current object in the Inspector using the normal print() statement formatting rather than ViewFrame's object formatting. This generally gives less detailed results, but gives you the option of displaying multiple levels of object structure at once (via the printDepth variable) – a feature which ViewFrame doesn't support in its own displays. The depth you specify with this command doesn't affect the printDepth setting for normal Inspector use.

### BACK BUTTON



Tapping this button returns you to the previous object in the object path. If there is only one object in the path, this button does nothing. The current object is lost, as far as ViewFrame is concerned (it remains in memory, if other references to it exist). The only way to get back to it is to reselect it in the previous object, or reevaluate the expression that created it.

### OBJECT PATH

...PreferencesForm.stepChildren[2]

This shows the sequence of expressions or selected slots that led up to the current object. If the list is too wide to fit in the ViewFrame window, text is dropped from the left end (representing older objects, which are generally less important than the newer objects towards the right), and an ellipsis appears at the left. The object path generally takes the form of a complete NewtonScript expression that would generate the current object.

There are some exceptions – slot names containing odd characters may not be enclosed in vertical bars, ViewFrame-specific expressions starting with a slash may appear, and Additions commands may put text here describing the action that they perform.

Tapping on the object path pops up a list showing all elements of the path, one per line. This is useful when the path is too wide to be entirely visible. Tap outside the list to dismiss it, or tap any item in the list to back up to that object. Items after the one selected are discarded, just as if you had tapped the Back button multiple times.

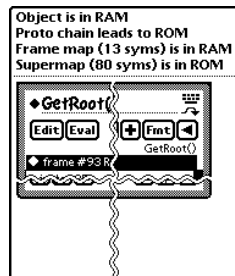
## OBJECT DESCRIPTION

◆ RO frame #5 Ref(5337017)

If you tap on the diamond to the left of the object description, a brief summary of the current object appears. See “Interpreting Results” on page 31 for more information on the data shown here.



If the current object is not an immediate value, tapping on the object-description diamond pops up some additional information about the current object. A few of the items add a related object to the object path – others do nothing if tapped. Many of the items make use of the new Newton 2.0 ability to exactly locate an object. In their descriptions below, *location* represents one of the following possibilities: “RAM”, “ROM”, “an invalid object”, or the name of an installed package.



### ONLY OBJECT IS IN LOCATION

Always shown under Newton 2.0. This is the location of the object itself.

### ONLY PROTO CHAIN LEADS TO LOCATION

For frames in RAM, this gives the location of the first item along the frame's \_proto chain that is not in RAM. If the current object is a view, this tells you which package contains the view template it was instantiated from.

### CLASSOF(CLASS) IS SYMBOL

For unusual objects whose class is not represented by a symbol, this shows the type of object its class actually is. Tapping on this item makes the object's class the new current object. This appears only in a few uncommon situations. For example, the class of a frame map object is an integer instead of a symbol.

### IS SUBCLASS OF SYMBOL

If the current object's class is defined as a subclass of another class, this item appears. For example, 'name', 'title', and 'address' are all subclasses of 'string'. Objects of those classes

have all the properties of strings, and can have all string manipulation functions applied to them, but may also have unique properties based on their particular class.

This item may appear multiple times if the object is a subclass of some class which is itself a subclass. For example, 'homePhone, 'faxPhone and various other phone number types are subclasses of 'phone, which is in turn a subclass of 'string. Newton 2.0 defines a new mechanism for defining subclasses by simply appending a period and additional text to the base class symbol. For example, '|nameRef.email| is automatically a subclass of 'nameRef under Newton 2.0, without any specific definition being needed. This item does *not* appear for subclasses of this type, since you can tell what the base class is by simply looking at the class symbol.

#### **ONLY** FRAME MAP (N SYMS) IS IN LOCATION

If the current object is a frame, and ViewFrame is able to locate its frame map, this item shows you the size and location of the frame map. This is not guaranteed to work, since there is no documented way of accessing frame maps. Tapping on this item makes the frame map object (an array of slot name symbols) the new current object.

#### **ONLY** SUPERMAP (N SYMS) IS IN LOCATION

Instead of having a single frame map with all of the slot symbols, a frame may have a partial map containing recently added slot symbols, which chains to a “supermap” (which was actually the frame’s map at some time in the past) containing the rest of the symbols. This can greatly reduce the memory requirements for frames. For example, if a frame with a map that’s in ROM has slots added to it, it isn’t necessary to copy the entire frame map into RAM in order to add the slot symbols. Instead, the frame gets a new map containing only the added symbols, which references the original ROM-based frame map as its supermap. Note that this may happen multiple times during the life of a frame, giving it a chain of supermaps. If ViewFrame is able to retrieve frame map info from a frame, this item appears once for each supermap in the chain, in reverse order of creation. In other words, the last appearing supermap item contains the oldest slots in the frame.

ViewFrame Additions can place additional items in the description pop-up. See the Additions documentation on disk for details on any Additions which do this, or for details on how to add your own items.

## OBJECT LISTING

This box shows a detailed description of the current object. See Interpreting Results on page 31 for more information on the data shown here.



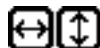
## SCROLL BAR

The scroll bar indicates the visible portion of the object listing relative to the entire listing. If the scroll bar is completely solid, the entire object listing is visible. If parts of the scroll bar are hollow, some of the object listing isn't currently visible. The solid part of the scroll bar is proportional to the visible area of the listing. The hollow parts at the top and bottom of the scroll bar are proportional to the unseen areas of the listing, above and below the visible area. You can drag the scroll bar to view a different portion of the object listing, or use the Down/Up buttons to scroll through it. Note that you can drag the scroll bar from any point, not just the solid part, which is sometimes too small to be easily dragged.




## DOWN/UP BUTTONS

Tapping these buttons scrolls the object listing in the indicated direction, by an amount slightly less than the height of the listing. If no further scrolling is possible in a particular direction, the corresponding button is hidden. Note that the standard Newton scroll arrows have no effect on ViewFrame – they are passed on through to the next-frontmost application that supports scrolling. You may find this somewhat confusing, but this lets you use ViewFrame for observing the effect of the scroll arrows on other applications.



## WIDTH/HEIGHT BUTTONS

Dragging these buttons resizes the ViewFrame window in the corresponding direction. You can also move the window around by dragging its border. The minimum window size is just wide enough to keep any buttons from overlapping, and high enough to display three lines of text in the object listing. The maximum size is the full size of the Newton screen. The window height is rounded off so that an integral number of lines of text fit in the object listing (unless the window is the full height of the screen, in which case there may be a partial line at the bottom or top of the listing).

 Resizing is now done by dragging an outline, rather than the ViewFrame window itself, making the process much faster. It is now possible to resize the window from all four sides: dragging all the way across the window begins resizing from the opposite side. When resizing vertically, the number of lines of text that fit in the object-listing area at the new size is shown.

## ZOOM BUTTON



Tapping this button switches the ViewFrame window between two default sizes and positions. The two default states are ViewFrame's initial size (somewhat larger than its minimum) and its maximum size. You can change either state as desired by moving or resizing the window while in that state.




## CLOSE BOX

This does exactly what close boxes normally do. ViewFrame attempts to preserve the object path when closed, and display the same current object when you reopen it. However, this cannot be guaranteed, since keeping hard references to objects in the path prevents the memory they occupy from being reclaimed by the NewtonScript garbage collector. If you reopen ViewFrame immediately after closing it, you will probably find that the entire object path is intact. If you reopen it later, after garbage collection has been performed, an earlier object in the path may become the current object if later objects were reclaimed. In the worst case, if there are no valid objects left in the path, ViewFrame simply displays a welcome message when opened.



## SHRINK BUTTON


 Tapping this button reduces ViewFrame to a tiny palette form, allowing easy access to other things on the screen that are obscured by the window. The palette can be dragged around the screen by its border, and can be closed or expanded back into the full ViewFrame window. It can also be expanded into a view selection tool called the ViewFinder. See "The ViewFinder" on the next page for more details.



Note that even if ViewFrame was closed while in the palette or ViewFinder form, it always opens with a full-size window.




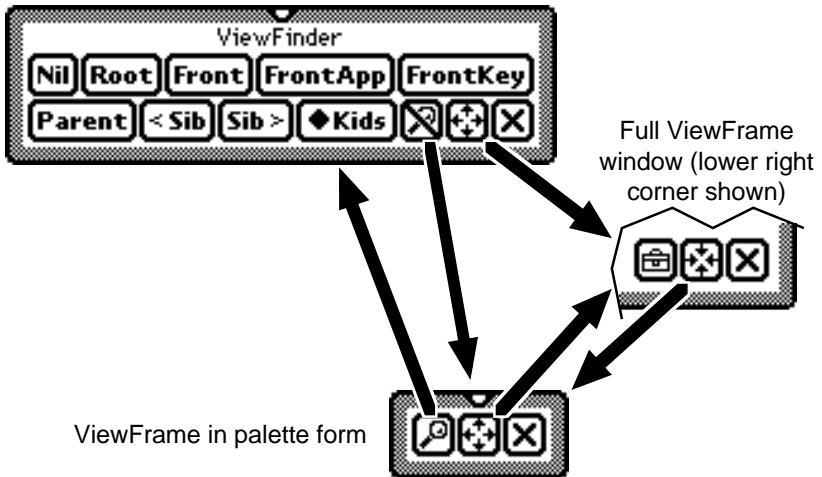
## TOOLKIT BUTTON

 Tapping this button opens or closes the Toolkit App (a component of the NTK which handles the Newton end of Inspector connections, package downloads, screen shots, and profiling) if it is installed. The Toolkit App appears in front of ViewFrame, even if it is an older version which is not normally a floating view.

In previous versions of ViewFrame, the area now occupied by this button and the Shrink button contained the Delay button, which closed ViewFrame and automatically reopened it five seconds later. The Delay button was commonly used to temporarily get ViewFrame out of the way, so that the Toolkit App could be used to open an Inspector connection. Hopefully you will agree that the new implementation, which gives you direct access to the Toolkit App, is more convenient. Other uses of the Delay button are now handled by shrinking ViewFrame and then expanding it when you're ready – no more racing against the five-second delay time.

## THE VIEWFINDER

 In addition to its normal full display, the ViewFrame window has two other sizes – a tiny palette which allows easy access to things that are obscured by the full window, and a view selection tool called ViewFinder. The transitions between these three forms are illustrated in the diagram below. Note that to go from the normal ViewFrame display to the ViewFinder, you first shrink the window to the palette form, then expand it into the ViewFinder.



### VIEWFINDER AND PALETTE USAGE

The ViewFinder lets you select any currently open view for further examination. The selected view is overlaid with a moving pattern of diagonal lines. Note that the selected view might not actually be visible, if it is obscured by other views. The selection lines cover the entire bounds of the view, including its border, and two pixels more on each side. This allows the selection to be seen even if the view has a height or width of zero, or is located just off of the screen. If the ViewFinder window partially obscures the selection, the selection appears behind ViewFinder. On the other hand, a completely obscured selection appears in front of ViewFinder, since you wouldn't be able to see it otherwise.

A description of the selected view object appears at the top of the ViewFinder window, in the same format as the object would appear in a normal ViewFrame slot listing. This is mainly useful for showing you the contents of a debug, text, or other identifying slot in the view. If no view is selected, “NIL” or “ViewFinder” appears at the top of the window.

The following buttons, some of which are found in ViewFinder only and some are shared between ViewFinder and the palette form of ViewFrame, are used to manipulate the view selection.

### THE NIL BUTTON



This button deselects any selected view. You should use it before doing anything that opens, closes, or moves any view other than the ViewFinder itself, since such changes may leave garbage (fragments of the selection lines) on the screen. If garbage is produced, tap on the Root button, then on Nil – this forces the entire screen to be redrawn.

### THE ROOT BUTTON



This button selects the root view, as returned by GetRoot().

### THE FRONT BUTTONS



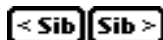
These buttons select the view returned by the GetView() function, when passed a parameter of 'viewFrontMost', 'viewFrontMostApp', or 'viewFrontKey', respectively.

### THE PARENT BUTTON



This button selects the parent of the currently selected view. It does nothing if no view or the root view is selected.

### THE SIBLING BUTTONS



These buttons move between children of the same parent view. A message is displayed if there is no previous or next sibling view.

### THE KIDS BUTTON



This button pops up a list of the children of the selected view: tapping on one selects it. If you can't tell which item corresponds to the desired child view, select any one then use the Sib buttons until the desired view is selected.

*ONLY* Child views of class 'clPictureView, containing an icon no more than sixteen pixels high, also show their icon in this pop-up. This makes it much easier to identify close boxes and other small picture buttons in the list.



### PALETTE TOGGLE BUTTONS

These buttons toggle between the ViewFinder and the palette form of ViewFrame.



### FULL-SIZE RESTORATION BUTTON


This button restores ViewFrame to its full size. If a view is selected in the ViewFinder, it becomes the new current object, completely replacing the existing object path. Otherwise, the current object and object path remain unchanged from when ViewFrame's Shrink button was tapped.



### THE CLOSE BUTTON

Closing ViewFinder or the palette is just the same as closing ViewFrame itself. When reopened from the Extras Drawer, ViewFrame always appears in its full window state.

## DRAG & DROP

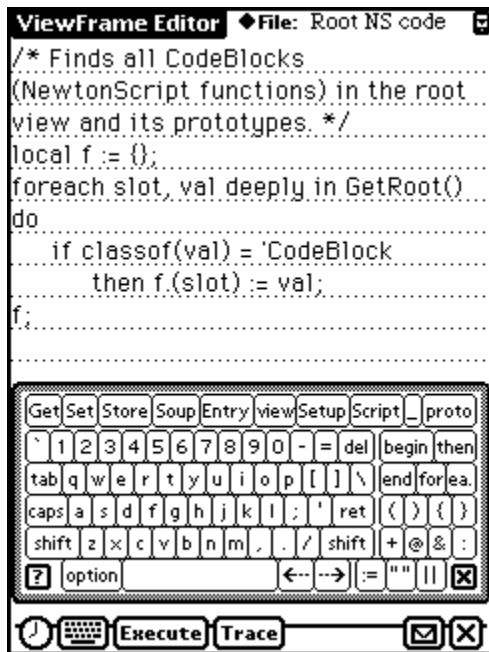
 ONLY Under Newton 2.0, both the ViewFinder and the palette form of ViewFrame accept dragged objects of several types, displaying the dropped object in ViewFrame. You may find the palette form to be more convenient for this purpose, since it accepts dropped data at any point. ViewFinder ignores drops on any of its buttons except the ones shared with the palette form. In either case, the entire window highlights when an acceptable dragged item is in a place where it can be dropped.

The drop types currently supported are icon, text, polygon, ink, picture, and meeting, listed in decreasing order of priority. Other types will probably be added in the future as Drag & Drop standards evolve. Dropped icons (from the Extras Drawer) are handled specially – ViewFrame attempts to display the base view of the application rather than the icon data itself, which is only really meaningful to the Extras Drawer. Obviously this won't work for icons that don't have an associated base view, such as auto part packages. In these cases you get the Packages soup entry for the icon, or the raw icon data.

If a single item is dropped, the item itself becomes the current object in ViewFrame. The description of the object in the object path indicates the item type: "Dropped text" for example. On the other hand, if multiple items are dropped, an array becomes the new current object, consisting of alternating type symbols and item frames. In this case, the object description reads "Multiple drops."

ViewFrame always tells the source of the dragged item that the item was *not* accepted, so that the original item doesn't disappear. However, not all sources of draggable items respect this. In particular, the current implementation of the Newton clipboard always deletes the original item after a drag. When dragging from the clipboard to ViewFrame, always start the drag with a double-tap so that a copy of the original item is made.

If you wish to test dragging of item types other than those supported, need more control over how the drop is accepted, or don't want the special handling of dropped icons to occur, the Drop Tester Addition is supplied. It is currently part of the VF+Dante package: see the Additions documentation on your disk for full details.



*THE VIEWFRAME EDITOR AND PROGRAMMER'S KEYBOARD*

# VIEWFRAME ACCESSORIES



## VIEWFRAME EDITOR

The ViewFrame Editor allows you to write simple NewtonScript programs directly on a Newton device. It is intended for experimenting with the NewtonScript language – it isn't suitable for writing complete applications. Even if you have the patience to tap in a full-scale program using the Editor, there currently isn't any way to turn it into a normal package that can be distributed.



### *USING THE EDITOR*

You open the Editor either by tapping its icon in the Extras Drawer, or by tapping the Edit button in ViewFrame. Usage is very similar to the Newton Notepad. The main differences are that the Editor doesn't accept drawn shapes, and that all entered text constitutes a single paragraph. Written text, regardless of location, appends to the existing text rather than starting a new text block. Scrolling is supported with the standard Newton scroll arrows. You should keep your program sizes to under 8K, although no limit is enforced.

Programs should consist of a series of NewtonScript expressions. There's no need to enclose them with begin and end, although it doesn't hurt to do so. Don't start the program with `func ()` as you do a script in the NTK. If you do, the result of executing the program is a compiled function object, not the result of evaluating that function. The displayed result is the value of the last statement executed, either the last statement in the program or a return statement.

Non-ASCII characters can be used in strings and character constants, even though this is not legal NewtonScript syntax: the Editor converts them to Unicode escape sequences as needed. This allows easy entry of programs containing non-English text, where constant use of Unicode would be rather inconvenient. Note that the use of non-ASCII characters anywhere else, such as in a variable name, is invalid and produces an error.

### *THE FILE SELECTOR*

### ◆ **File:** Root NS code

The name of the current program is shown at the top of the screen. Tapping it (or the Overview dot between the scroll arrows) pops up a list of all Editor programs. Tapping on the name of one of the programs switches to it. Note that what the Editor stores is the text of your programs, not the result of compiling or executing them.



### THE STATUS ICON

The status icon is located at the top right-hand corner of the screen. It appears as a Newton with an arrow pointing down, if the current program is stored in internal memory; or as a card with an arrow pointing up if the program is stored on a card. Tapping on the icon shows the version number of the Editor and the amount of storage space taken by the current program.



### THE KEYBOARD BUTTON

This control displays an on-screen keyboard. The Programmer's Keyboard is used if it is installed, otherwise the standard Newton keyboard appears. You may need to tap in the editing area to get the insertion point to appear. If you have the Programmer's Keyboard installed but would like to use the standard keyboard, you can bring it up by double-tapping in the editing area.



### THE EXECUTE BUTTON

This button compiles and executes your program. Simple results, currently defined as integers, reals, and textual error messages, are displayed in a standard notification dialog box. Other result types are displayed using ViewFrame, so that you can use its various object-viewing options to look at the result.

If you would like to view an integer or real result in ViewFrame, store it in a slot in a frame or array, then return that object instead. If a simple result is generated while the notification box is already visible, the new result isn't immediately displayed – you must tap the down arrow to scroll to the new item. Likewise, you can tap the up arrow while the notification box is open to scroll through the last few results or other messages.



### THE TRACE BUTTON

This button does the same thing as the Execute button, except that function and slot access tracing are turned on (by using “trace := true”) during the process. Your Newton device must be connected to the Inspector for this to do anything. There is some irrelevant trace output both before and after the execution of your program, since changes to the trace state don't take place immediately. To help locate the useful portion of the Inspector output, the actual execution is marked off by the strings `*** EXECUTION BEGINS ***` and `*** EXECUTION ENDS ***`.



### THE ACTION BUTTON

Tapping this presents a list of options for sending or otherwise acting on the current program. (The options may be slightly different than those shown on the graphic on the next page under Newton 2.0). It works much the same as it does in most other Newton programs.



**PRINT PROGRAM /FAX**

This command produces hard copy of the current program. The output is currently limited to a single page, since large programs aren't really appropriate for the Editor.

**BEAM**

This sends a copy of the current program to another Newton via IR. The other Newton must also have the Editor installed to allow the beam to be put away.

**MAIL**

The mail command mails a copy of the current program via NewtonMail. If sent directly to another NewtonMail account, the receiver (who must also have the Editor installed) can simply put the item away. In this case, the text of the message can be scrubbed out before sending since it is not used.

Programs can also be sent to people on other systems, via Internet. In this case, the message must be sent Text Only. The text of the message contains the program, wrapped in NewtonScript statements that install it in a Newton. The recipient should paste the text into the Inspector window on his desktop computer, then select and execute it. Instructions for doing this are included as part of the message. Note that any non-ASCII characters are lost in the transfer, and that the program may not be very readable in the message: quotes and backslashes require some translation to turn the program into a legal NewtonScript string constant.

**>INSPECTOR**

This dumps the current program to the Inspector in the same wrapped format used when mailing. This can be used by a person with no NewtonMail account to mail a program by copying the program from the Inspector window and pasting it into a message being sent in some other way. This is also useful for editing a program in the large-screen environment of the Inspector, then sending it back to the same Newton. Note that each time you execute the message to send it to the Newton, a new copy of the program is made. You need to manually delete older copies if desired.

**NEW PROGRAM**

This command creates a new, blank program, and makes it the current program.

**RENAME**

Rename opens a slip allowing you to specify a new name for the current program.



**DELETE**

This command trashes the current program. This action is undo-able.

**Duplicate**

Duplicate makes a copy of the current file and opens the Rename slip to allow you to give it a new name. Note that if you close this slip without entering a new name (or by various other methods), it is possible to create multiple programs with the same name. This causes some cosmetic problems in the file selector, but is harmless.

**MOVE TO/FROM CARD**

This command does what it says. The status icon in the upper right-hand corner changes to reflect the new location of the program.

*EDITOR PROGRAM EXECUTION*

If an error occurs during compilation, and the error message specifies a line number, the insertion point is moved to the end of that line (you must have a keyboard open for this to be visible). The actual error is often on the previous line, and can be anywhere prior to the indicated position.

Execution of your program takes place in the view context of the Execute button even if you run it by tapping Trace instead. Compare this to code entered in the Inspector, which is executed in the context of the global variables frame, a rather uncommon situation in normal Newton programming. The main difference this makes is that you don't have to use `GetRoot()` to access items particularly view methods in the root view, since it is in the inheritance path of your program. It is possible for statements you enter such as `"self:close()"` to render the Execute button unusable. If this happens, simply close the Editor and reopen it.

*SAMPLE PROGRAMS*

Here are some sample programs to try in the Editor. You can avoid typing them in yourself by installing the VF Ed SoupMaker package, which adds these and other samples to the Editor.

```
for i := 0 to 369 do begin
  local x := compile($@ & i);
  x := call x with ();
  if classof(x) = 'frame and
    x.sndFrameType exists
  then begin
    playsoundsync(x);
    sleep(30);
  end;
end
```

This program loops through all magic pointers which are defined under Newton 1.x (which is done by generating an expression referring to each one and then compiling it), identifies those which are sounds, and plays them. This program can be easily modified to search the magic pointers for any other specific type of object.

```
local f := {};
foreach slot, val deeply in GetRoot() do
  if classof(val) = 'CodeBlock or
    classof(val) = '_function
  then f.(slot) := val;
f;
```

This program lists all of the NewtonScript byte-code functions (either Newton 1.x or 2.0 format) in the root view and its prototypes. Note the use of a local variable to accumulate found items, since only a single result object can be returned from your program. This can easily be modified to search elsewhere or for a different object type.

```
BuildContext({
  _proto: @180,
  viewBounds: SetBounds(0, 10, 225, 225),
  stepChildren: [{
    viewClass: 76,
    viewFlags: 1,
    viewBounds: SetBounds(5, 5, 209, 209),
    icon: @320 }]
}):Open()
```

This sample demonstrates creation of a new view from a program. Note the use of numeric values such as @180 and 76, rather than the equivalent constants, `protoFloatN-Go` and `dPictureView`. This is necessary since these constants exist only in the NTK, not in the Newton software.

### EDITOR INTERNALS

There are some defined items in the Editor's base view that may be of use to advanced programmers:

- To get a reference to the base view:

```
VFEd := GetRoot().|VFEditor:JRH|;
```

- To specify a different font for displaying/printing programs:

```
VFEd.EditorFont := { /* font frame */};
```

This change should be made when the Editor is closed, and lasts until a reset.

- To add a new program to the Editor from another application:

```
VFEd?:AddFile(file)
```

where *file* is a frame containing two slots:

<code>title</code>	a string containing the name of the program.
<code>text</code>	a string containing the program text.

This call must be made when the Editor is open. This does not overwrite any existing programs, even if they have the same name.

The Editor keeps its programs in a union soup named “VFEditor:JRH”. This soup remains, even if the Editor itself is deleted. If you would like to completely remove the Editor from your system, you need to manually delete the soup, using a soup utility like Ragout, from Creative Digital, or direct soup calls from ViewFrame.

## PROGRAMMER'S KEYBOARD



The Programmer's Keyboard is an on-screen keyboard designed to simplify the entry of NewtonScript expressions. It supports all the features of the standard on-screen keyboard except for entry of new words into the user dictionary, which generally isn't a good idea when programming. Additionally, the Keyboard has many common NewtonScript keywords and symbols available for entry with a single tap, and can be switched to a Dvorak keyboard layout for those of you who prefer it.

The Keyboard can be directly accessed from ViewFrame or the ViewFrame Editor by tapping the keyboard button. You can leave it open, or open it yourself from the Extras Drawer, and use it in other Newton applications. In most cases, you first need to tap in the area where you want to enter text, to make sure the insertion point appears there.



Tapping on the question mark in the lower left corner of the Keyboard pops up a list of options:

- QWERTY or Dvorak layout. Currently, your layout setting isn't permanently stored, so the Keyboard reverts to QWERTY layout every time your Newton device is reset.
- Unicode entry mode. When this option is enabled, a row of additional keys appears above the main keyboard to allow you to type in an arbitrary Unicode character via its four-digit hexadecimal value. After the third digit is entered, the additional keys change to the actual character that is produced by each possible fourth digit. Additional taps are interpreted as different fourth digits, with the first three digits of the Unicode value staying the same. At any time, the **clr** key can be tapped to restart entry from the first digit.

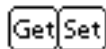


*THE PROGRAMMER'S KEYBOARD WITH UNICODE ENTRY MODE ENABLED.*

- Select Unicode font. This option pops up a list of installed fonts: selecting one causes it to be used for the Unicode character display in the additional row of keys. This allows you to see how the characters differ between fonts. Note that this option has no effect on the font that Keyboard-generated text is actually displayed in: that is controlled by the field the text is being typed into, and possibly modified by the Styles slip or similar utility.

Most of the keyboard keys are self-explanatory. A few need some explanation.

#### *GET, SET BUTTONS*



These buttons actually enter the text “Get()” and “Set()” then back up the cursor two spaces so that you can finish entering the function name. Tap the right arrow once to begin entering parameters, and again to continue with normal expression entry.

#### *" ", | | BUTTONS*



These enter a pair of punctuation marks, and leave the cursor between them so you can enter a string or symbol. Tap the right arrow once you're done. If you need only one quote mark or vertical bar, tap Shift and then the character, as you would with the normal keyboard.

---

## INTERPRETING RESULTS

### OBJECT DESCRIPTIONS

The brief object summary, displayed just above the box containing the detailed object listing, contains the following information for pointer references (frames, arrays, and binary objects):

- RO, if the object is read-only (either because it's in ROM, or because it's part of a package, which isn't writable once installed).
- The object's `PrimClass` (Frame, Array, or Binary), followed by its class, if not the default. Example: `Array:pathExpr`.
- The length of the object, preceded by a “#” (pound sign). The length is measured in slots (for frames), elements (for arrays), or bytes (for binary objects).
- The object's reference value, shown as `Ref(nnn)`. About the only use for this is to type it into the Inspector to view the same object there. Note that there are easier ways of doing this (the Inspector options in the Action menu), and it's not guaranteed to work. (See *NewtonScript Secrets* on page 48 for a discussion of reference values.)
- If the object is referenced via a magic pointer, it's shown in the form `@nnn`. You can look this up in the Definitions file (which comes with the NTK) to see what the object's name is.

Immediate values have simpler descriptions, since much of the information described above is meaningless for them:

- The object's class: possibilities are `Int`, `Char`, `Boolean`, and `Weird_Immediate` (NIL has this class). Newton 1.x systems also have a `Mark` class, but it's never used.
- The object's reference value, as mentioned above.

### OBJECT LISTINGS

`ViewFrame` has a variety of viewers for specific object types, plus generic viewers for the basic `NewtonScript` object types if no specific viewer is available. Most of these viewers build their displays from just three different components – text blocks, labels, and slots.

Text blocks are displayed with the Fancy (New York) font. They are used for multi-line text items, such as strings and function listings. You can select text from these items and move it elsewhere, just as you can with most text on the Newton screen – hold down the pen until you hear a squeak and the ink becomes very thick, then mark through the text to be selected.

Labels are displayed with the bold Simple (Geneva) font. Text from these items can be selected and moved as well, but only one line at a time (since each line of a label is a separate item).

Slots are displayed with the plain Simple font. Tapping one of these items at most points makes the slot the new current object. However, the extreme left end of a slot item doesn't accept taps. Instead, you can make a gesture there. Three gestures are supported: selection, scrubbing, and horizontal lines.

Selection works the same here as anywhere else, except that you can only select text starting at the left end of the item. This is mainly useful to pick up the name of a slot, for use in an expression that requires it. The other two gestures produce an expression in the entry line. Scrubbing out a slot produces the expression `"/RemoveSlot(%,'slotname')"`; tapping Eval then deletes the slot. Drawing a horizontal line produces an assignment statement to change the value of the slot: `“.slotname :=”` for slots in frames, `“[index] :=”` for array elements. As a special case, drawing a line through a slot in an open view produces a `SetValue()` expression that changes the slot's value and updates the view. Write or type in a new value for the slot, and tap Eval to make the change.

### *SOUP ENTRIES*

Regardless of the object display format used, any object which is a soup entry has a four-line header at the top of its listing, similar to:

```
Soup entry
• In soup: System
• On store: Internal
907 bytes (461 text)
```

Tapping on the second or third line makes the soup or store the new current object. This allows you to get info about (or send messages to) those objects. The numbers on the fourth line are the results of applying the `EntrySize()` and `EntryTextSize()` functions to the object. Dirty appears at the start if the entry has unsaved changes.

Before attempting to display an object itself, ViewFrame gives all installed Additions a chance to supply a custom display (which may either precede or replace the normal display). The object types ViewFrame currently has built-in viewers for are:

### *NIL AND TRUE*

These aren't particularly interesting objects, but they have to be handled as special cases since NewtonScript has no built-in text representation for them.



ICONS. EXAMPLE EXPRESSION: @330

ViewFrame shows the dimensions of the icon plus the icon image. Icons with a mask show the mask plus the icon as it appears when highlighted. Note that the highlighted view currently doesn't work in printed or faxed output – the plain icon appears instead.

PICTURES. EXAMPLE EXPRESSION: @320

ViewFrame shows the size of the picture and the actual picture.

**ONLY** SHAPES. EXAMPLE EXPRESSION: MAKEOVAL(0,0,50,20)

Shape objects (as determined by IsPrimShape()) are displayed with the same viewer used for pictures. Shape arrays may also be displayed this way, but ViewFrame cannot reliably identify them – any element of the array might actually be a style frame. The system provides no easy way of detecting them.

SOUNDS. EXAMPLE EXPRESSION: @102

ViewFrame plays the sound, and shows its length in bytes, sampling rate in kilohertz, and duration in seconds. Tap the Fmt button, then reselect Normal format, to play the sound again.

INTEGERS. EXAMPLE EXPRESSION: TIME()

Integers values appear in decimal and hexadecimal, and are displayed as time and date values if in an appropriate range (approximately 1942 to 2094). Also, symbolic constants can be shown for several common view slots that have integer values. This is done automatically if you tap on a slot with an appropriate name, such as viewFlags. For integers viewed by other means, such as writing them into the expression entry area, you must tap on “View as:” in the listing, and select the desired format from the pop-up list.

CHARACTERS. EXAMPLE EXPRESSION: \$X

ViewFrame shows the character and its Unicode value in decimal and hexadecimal. Note that several commonly used characters (such as the checkmark, and the diamond used to indicate a pop-up menu) exist only in the system (Espy Sans) font – these appear as a box in the listing, which uses a different font.

STRINGS. EXAMPLE EXPRESSION: @115

ViewFrame shows the string. Previous versions indicated carriage returns with a “↵” symbol. However, Newton 2.0 units no longer have that symbol in the font used for string display, so its use has been dropped. This change also saves memory, since a copy of the string no longer has to be made.

ANYTHING ELSE, THAT HAS A PRINTABLE REPRESENTATION

If `SPrintObject()` returns a non-empty string for the current object, this text is used for the listing. This handles various other object types, such as symbols and reals.

### *GENERIC VIEWERS*

Any object which doesn't match any of the types listed above (or has been set to use the Alternate format) is displayed with a generic viewer, based on its `PrimClass`.

### IMMEDIATES

“(no printable representation)” is shown, for lack of anything better to do with such an object. The reference value shown in the object description is the only information you have about the object. This situation should occur very rarely.

### BINARY OBJECTS

These are displayed in hexadecimal. The listing shows eight bytes of data per line, and has four columns, although you may not be able to see the right-most columns if the ViewFrame window is narrow. The left-most column is the offset of the current line in the object. It is shown only on every other line, to reduce the amount of display data generated. The next column shows the eight byte values for this line. The third column shows the ASCII text equivalents of the same eight bytes. The right-most column shows a Unicode text equivalent for the four pairs of bytes on this line. This column is not entirely visible on Newton devices with the standard screen width of 240 pixels. Currently, it can only be fully seen on a Newton 2.0 device with the screen rotated.





### FRAMES AND ARRAYS

A generic slot listing is available for frames and arrays that don't have a specific viewer. This is probably the most common display format you see in ViewFrame, since views, view templates, and many other common application data structures fall into this category. Note that if the Alternate display format is selected, frames are viewed deeply (including slots contained in the frame's prototypes).

## **GENERIC FRAME/ARRAY LISTINGS**

Each line in this type of listing represents one slot in the current object, and can be tapped to display the contents of that slot as the new current object. Each line has the name of the slot (a symbol in frames or an integer in arrays), a colon, and a brief textual description of the slot's value. If the current object is a frame which is being viewed deeply (by selecting the Alternate format from the `Fmt` button), there may be some dashes to the left of the slot name. Slots contained directly in the current object have

nothing added. Slots in the object's `_proto` have one dash, slots in the `_proto's _proto` have two dashes, and so on. The slot value descriptions have one of the following forms:

- SELF, if the slot refers to the current object. Tapping such a slot does nothing, since it won't show you anything you aren't already looking at. Note that this use of SELF is not quite the same as what it means in a NewtonScript program.
- NIL or TRUE.
-  For all of the following data types, if the slot is a magic pointer reference this is indicated at the start of the displayed description, in the form `@mmm`. Previously, this only worked for certain object types: strings and functions referenced by magic pointers weren't properly identified.
- Symbols, preceded by a single quote.
- Strings, enclosed in quote marks.
-  Rich strings (strings which include ink as well as text) appear as quoted strings with a dollar sign in front. Note that strings which have a null character stored in the middle may be incorrectly identified as rich strings.
-  Subclasses of the 'string class are now recognized, and displayed with the class symbol and a colon in front of the string data (which may be either a normal or rich string).
- Reals.
- Characters, preceded by a dollar sign.
- Integers, in decimal, with a symbolic constant equivalent if the value can be expressed with a single constant. Combinations of constants (such as `vApplication + vClipping`) are not decoded here – tap on the slot for a more detailed listing.
-  As a special case, integer values in any slot named “debug” are interpreted as encoded versions of the debug strings that formerly identified many system prototypes. Unfortunately, most debug strings have been removed on recent Newton devices, to save ROM space. If Apple's `DebugHashToName` package is installed and is able to translate a given debug integer, the equivalent debug string appears in parentheses after the integer's value.
- Bounds frames, showing each of the four slots. Example: `{L0,T13,R240,B302}`.
- Various types of executable function objects, showing the number of expected parameters in parentheses. The possibilities are:

<code>funch(#n)</code>	ordinary Newton 1.x byte-code function;
<code>CFunction(#n)</code>	Newton 1.x built-in function;
<code>BinCFunc(#n)</code>	Newton 1.x native function;
<code>newfunc(#n)</code>	Newton 2.0 bytecode function; and
<code>newCfunc(#n)</code>	Newton 2.0 built-in or native function.

- **ONLY** The Newton 2.0 function formats are only recognized under Newton 2.0 – a 1.x system displays them as frames with an unusual class.
- As a special case of the above, intercept functions installed by the VF+Intercept Addition are recognized and shown as “intercept(#n)”, whether they are in Newton 1.x or 2.0 format.
- For any other types of binary objects, the object’s class and length are shown in angle brackets. Example: <dictdata:#207>.
- For arrays, the object’s class (if other than the default) and length are shown in square brackets. Examples: [#8], [pathExpr:#3].
- For frames other than those listed as having special representations, the description is the same as for arrays, except that curly brackets are used instead of square brackets. Additionally, one of the following informational items may be present at the end:
  - GetRoot() or GetGlobals(), if the frame is the same as the object returned by either of these functions.
  - The frame’s debug slot, if it has a string value.
  - **DEBUG** A debug slot with an integer value, if the DebugHashToName package is installed and can translate the integer into a debug string.
  - The value of various identifying slots (text, tag, name, title, overview, preallocatedContext) if present, labeled with the slot name. Exceptions: the overview slot is abbreviated as “ov”, and the preallocatedContext slot is labeled “declAs” since it gives the name with which a view was declared.
  - **EXE** If the frame has none of the identifying features listed above, its \_proto chain is searched for a frame that is identifiable. Any descriptive text generated in this way is preceded by “>”s, one for each \_proto link followed. If no description is found, ViewFrame tries to find a frame referenced by a magic pointer in the \_proto chain. If found, the reference will be displayed in the @nnn form, again preceded by “>”s to indicate the depth at which it was found.


Example:     @212 {#9} protoSlider

This indicates a frame with nine slots, including a debug slot with the value protoSlider, which is referenced by the magic pointer @212. You can look up @212 in the NTK Definitions file to find the proper name of this object – protoSlider. This object identified itself, but not all magic pointer objects contain a debug slot.

- Any other objects that have a text representation (generated by the SPrintObject() function) use this as their description.

- Any remaining objects are described by their PrimClass, class, and reference value. This should happen very rarely.

## LOCATION LISTINGS

 The Location format, which can be selected from the Fmt pop-up, requires Newton 2.0 and does not apply to immediate values (they are displayed the same as in Normal format). The resulting display is similar to the generic listing for the object, but the emphasis is on where the parts of the object are stored, rather than the values they have. In the description below, *location* represents one of the following possibilities: “RAM”, “ROM”, “an invalid object”, the name of an installed package, or “immediate” (indicating a frame or array slot that is self-contained, not referring to any other object).

For all object types, the Location listing starts with “Object is in *location*”, referring to the body of the object. Arrays and binary objects then locate their class symbol, which could be in a different place than the body of the object: “Class sym is in *location*.” Frames don’t need this line, since their class symbol is either contained in an ordinary slot named class, or is the default value of 'frame which is built into the system.

For binary objects, this is the end of the listing – the object body and class symbol are all that there is. Arrays continue with a listing of the locations of each of their elements. Each line is of the form “*index: location (value)*”.

Frames appear in one of two forms, depending on whether ViewFrame is able to locate the frame’s map (there is unfortunately no documented way of doing this). If the attempt fails, the listing says “No frame map info available”, and simply contain two lines of data for each slot in the frame. The first line in each pair concerns the slot name symbol: “*symbol: sym in location*”. Tapping on this line makes the symbol itself the new current object. The second line concerns the slot value: “*val in location (value)*”. Tapping on it makes the slot value the new current object, just as in a normal frame listing.

Note that even if frame map info is not available, it is still possible to make educated guesses about the frame map’s location, based on the locations of the slot symbols. If any slot symbols are contained in a package, it’s likely that at least part of the frame map is in that package as well - in other words, the frame would become invalid if that package was removed. If any slot symbols are in RAM, or there are symbols in both ROM and a package, it’s guaranteed that at least part of the frame map is in RAM.

If the frame’s map can be located, the listing is broken into blocks, one for each component of the frame map. The first block starts with “Frame map is in *location*”, and contains symbol/value lines as described above for each slot described in the frame map object itself (which includes at least the most recently added slot names, if not all slots). It is possible for a frame map to be incomplete, and refer to a “supermap” (actually, a previous version of the frame’s map) that contains earlier slot names. If this is the case, there is a second block starting with “Supermap is in *location*”, containing the symbol/

value lines for the slots described in the supermap. Supermaps may themselves have supermaps, resulting in additional blocks, although it's fairly uncommon to see a frame with more than one level of supermap. The lines containing the frame map or supermap locations can be tapped on to make the map object itself the new current object.

## FUNCTION LISTINGS

If you have the VF+Function Addition package installed, it can reconstruct the original source code of compiled NewtonScript functions with fairly high accuracy. This is done automatically whenever a bytecode function object is viewed. This is an inherently inexact process, but it is possible to a far greater extent with NewtonScript than with almost any other compiled language – even the names of local variables may be recoverable from the compiled function. Some of the unavoidable limitations on function listings are:

- All original source code comments are lost. However, VF+Function may generate comments of its own. Magic pointer references are followed by a comment giving a brief description of the referenced object, to make up for the fact that the proper names for magic pointer objects do not exist in the Newton.
- The exact usage of parentheses and begin...end blocks is lost. VF+Function generates the minimum number of these grouping constructs needed to unambiguously indicate the order of operations.
- There are several instances where two seemingly different expressions produce identical compiled code. Example: A and B, and if A then B are indistinguishable if the value of the expression is actually used (which is always true for the last expression in a function, which yields the function's return value). VF+Function tries to guess which interpretation makes the most sense. In the example above, if B is a compound statement, the if...then interpretation is chosen. Its guesses are sometimes unavoidably wrong, producing code that looks rather strange (but functions exactly the same as the programmer originally intended).

Some confusion can result when listing a function compiled by the NTK that includes constants that are defined elsewhere (via the constant keyword or the DefConst() function). The resulting compiled code is exactly the same as if the constant object had been written as a literal in the function. It is possible, however, to define a constant as an object for which no syntax exists for writing it directly.

For example, consider this constant definition:

```
DefConst('kMyPatt', SetClass( "\u8844221188442211", 'pattern'));
```

This creates a constant defining a fill pattern which can then be used in a function. However, there is no way to list such a function in pure NewtonScript form, since NewtonScript has no syntax for directly creating a binary object of arbitrary class. Here are some of the ways in which constants can be used to escape the bounds of pure NewtonScript, and how VF+Function deals with them:

1. A single constant can be used several times in the same function, in which case all uses refer to the same object. Assuming that the constant's value is expressible in NewtonScript, VF+Function writes out its value in full everywhere it is used. There's no easy way to tell if identical literals in a function all refer to the same constant object, or whether they are separate objects.
2. A constant can be used to embed a literal that can't be expressed directly in NewtonScript, such as the binary object of class 'pattern in the example above. In this case, VF+Function precedes the function listing with the heading "External objects used:" followed by a list of such objects, with arbitrary names of the form `kObjectn` assigned to them. The objects can be tapped on to see their values, just as in a normal frame listing. Each use of the constant in the function listing appears as a string literal containing the arbitrary assigned name of the object.
3. A constant can be used to define a literal function that can be called directly, without any inheritance search. This is subtly different from the normal NewtonScript ability to nest a function inside of another: a nested function has a context which includes the enclosing function's context and its local variables, while a constant function has no context at all. VF+Function lists nested functions using the normal NewtonScript syntax for them. Constant functions are assigned arbitrary names of the form `kFuncn`, and appear in the "External objects used:" section as described in case number two.
4. The NTK's Platform file defines various library functions which can be used in programs. These are simply constant functions as described in case number three, however VF+Function can recognize many of them, and display them by their proper name. Recognized library functions are displayed in the same form that they are written in the NTK: with a "k" in front of and "func" after the actual function name. Unrecognized functions (such as the Alarm Library routines) appear with a `kFuncn` name as in case number three. Many library functions are recognized via a generic mechanism that should also work with many future library functions, with no needed changes to ViewFrame or VF+Function.

## TIPS & TECHNIQUES

### FINDING AN APPLICATION

In order to examine an application with ViewFrame, you first have to find it. Version 1.2 introduces two new ways to do this: dragging icons from the Extras Drawer (Newton 2.0 only), and the ViewFinder. Several Additions are supplied to help with this task: Get App in the VF+General package (Newton 1.x), Get Part and Get Base View in the VF+Dante package (Newton 2.0). Here are some of the other paths you can take to reach a particular application:

#### GETROOT()

All applications, and many other system components, exist in the root view. However, this frame is too big to easily find things in it, although sorting it by slot helps.

#### GETROOT().|MYPROGRAM:MSYIG| (EXAMPLE)

If you know an application's symbol, you can simply enter it, either while viewing the root view, or as a single expression starting with GetRoot(). Note that application symbols normally include punctuation marks, so you have to enclose them in vertical bars.

#### DEBUG("name")

If the "Compile For Debugging" option is enabled for your program in the NTK, you can find any named view with this expression. Note that only the base view can be found this way if the application is closed. Other views can only be found if they are open or declared in an open view.

#### GETGLOBALS().EXTRAS

In Newton 1.x only, this array contains information about all installed programs. Each array entry includes a text slot giving the program's name, so it's easy to find the entry corresponding to a program of interest. The important slots in entries for installed packages are formContext, which is the application's base view, and theForm, its base template. The entries for built-in programs unfortunately do not have any direct reference to the program itself: you must tap the app slot to get its symbol, then use "/GetRoot().%" from the expression pop-up list to look up the symbol in the root view.

#### GETROOT().EXTRASDRAWER.VIEWCHILDREN

This array is closely related to the one above, and also works in Newton 1.x only. It contains the actual view templates for everything that appears in the Extras Drawer. Unfortunately, it doesn't include entries for applications beyond those that fit in the



Extras Drawer (30 total on Newton 1.x devices), and doesn't include direct references to the applications, so you always have to use `"/GetRoot().%"` to go any further.

`GETVIEW('VIEWFRONTMOST)`

If you have an application open behind `ViewFrame`, this expression often finds it. Note that the found view may not be the application's base view. To get to the base view, keep tapping on the `_Parent` slot in views until you reach one that shows `GetRoot()` after its `_Parent` slot. Using the symbol `'viewFrontMostApp` works much the same. In either case, the application must have the `vApplication` bit set in its `viewFlags` in order to be found. `'viewFrontKey` generally just finds the expression entry area in `ViewFrame` itself, but could be used to find another application if you tap in an input area of that application just before tapping `Eval` in `ViewFrame`.

`GETROOT():CHILDVIEWFRAMES()`

If all else fails, open the desired program, close everything else possible, and try this expression, which gives you the base views of all open programs. Some trial and error may be needed to determine which view is the one you want.

## WHAT AM I LOOKING AT?

Any frame that you find with a `viewCObject` slot is an actual view. A `NIL` value indicates a closed view; a numeric value means that the view is open (but possibly hidden). No meaning can be extracted from the exact numeric value: it's a reference to an object outside of the `NewtonScript` environment. Anything that you reach via `_proto` slots from a view, or find in a `viewChildren` or `stepChildren` array, is a view template of some sort. The first level is the view's template as laid out in the `NTK`, or possibly a template created at runtime. Subsequent levels of prototypes may be user `protos` that the view is based on.

Once you reach a frame referenced by a magic pointer (a `@nmn` notation appears in the `_proto` slot and in the description of the referenced object), you have left the program's templates, and are looking at a built-in `proto` template. There may be multiple levels of prototypes in the `proto` template, depending on its complexity. Eventually, you reach a frame that has no `_proto` slot, but does have a `viewClass` slot – this determines the fundamental characteristics of the view.

When looking at an open view or its first-level template, you can locate it on the screen (if not obscured by other views) by using the `"/DV(%)"` expression from the `pop-up` list. The main effect of the `DV` function is to dump information about the view and its children to the `Inspector`. It also flashes the view several times, which is useful even if the `Inspector` isn't connected.

## BROWSING SOUPS

There are several Newton utilities, such as Ragout from Creative Digital, available for examining and modifying the contents of soups. You can use ViewFrame for such purposes as well. This is much easier if you have an Addition package installed that adds soup features (see the Additions documentation on your disk for details), but it can be done using only the basic ViewFrame package by entering appropriate expressions. Many of the needed expressions appear in the pop-up list, so usually the only typing required is the name of a soup.

Here's a general procedure for browsing entries in a soup:

- Select the “GetUnionSoup(“ ”)” expression from the list, type in the name of the desired soup, and tap Eval to get a soup reference. Note that you can use a UnionSoup even if the desired soup exists only on one store, such as the Newton 1.x Inbox and Outbox soups. Or:
- If you'd prefer to examine entries only on a particular store, start with GetStores() instead. Select the desired store from the resulting array (element 0 is always the internal store), then select the “:GetSoup(“ ”)” expression and type in the desired soup name.
- Now that you have a soup reference, use the expression “/Query(%,{type:'index'})” to get a cursor (the Query call is slightly different under Newton 2.0). This is the simplest query possible. You may want to add additional slots to the query spec to sort the results or retrieve only specific entries.
- If the total size of the soup entries is small enough, you can apply the “/MapCursor(% ,nil)” expression to the cursor to get an array of all entries in the soup. You can then browse through the entries using ViewFrame's normal object viewing features – no more soup-related expressions need be entered. However, it is quite possible that MapCursor will generate an error if the soup is too large to fit entirely in the frames heap. In that case:
- It is also possible to browse the soup entries one at a time. To get the first entry use the expression “:Entry()” on the cursor. To retrieve subsequent entries, back up to the cursor and use “:Next()”. You can also go back to previous entries by using “:Prev()”. Don't forget to back up to the cursor object before using either Next or Prev. If you apply one of these methods to an entry rather than a cursor, you get an error.

## UNABLE TO BUILD LIST


If this notice appears in the object display area, an error occurred while ViewFrame was generating the display items for the current object. This is usually due to a lack of memory. The descriptions of frames or arrays with hundreds of entries simply won't fit in the frames heap of current Newton models.

Another possibility is that you are trying to view a frame or array with an invalid slot, which is a reference to a package that is no longer installed. This often occurs in the GetGlobals() frame, which is where variables assigned from the Inspector are located. For example, if you enter an expression like this in the Inspector:

```
x := debug( "MyApp" ) . _proto
```

and the referenced application is later removed or replaced with a newer version, the variable *x* becomes invalid. Any attempt to display the frame containing the variable generates an error.

There is not much you can do about this problem in ViewFrame, unless you know the name of the variable causing the problem. You can connect to the Inspector and display the frame there, since it is capable of continuing display after an error. Note which slots have an error message as their value, and use RemoveSlot on them (or set their value to NIL). Or you can just reset your Newton device, which rebuilds the globals frame from scratch.

Another possibility, hopefully not a very likely one, is that the “UNABLE TO BUILD LIST” notice indicates a bug in ViewFrame.  You can tap on the notice to see the exception data generated by the error, without having to be connected to the Inspector. However, additional information about the error may be displayed only to the Inspector. If you think you've found a bug, please let us know. You need to tell us the exact object you were trying to view, as well as all Inspector output produced during the attempted display, in order for us to duplicate and fix the problem.

Even if the notice appears, the object that failed to display is still the current object. You can select slots from it and perform any other operations on it by entering expressions or with Addition commands, just as always. Selecting the Alternate format from the Fmt button may work even though the Normal format doesn't. The Inspector print option in the Action menu still works. If the object is a binary object viewed in hexadecimal, the >Inspector option works. However, the >Inspector option on other object types, as well printing and faxing, simply outputs a copy of the notice.

## ADVANCED USES

### DIRECTLY USING VIEWFRAME

It's possible for Newton programs that you write to use ViewFrame to display objects, much as you might use `print()` to display objects in the Inspector. This is especially useful when developing programs that use the serial port. On current Newton devices with only one port, you can't have the Inspector connected at the same time your program is using the serial port for its purposes.

In order to use many of the techniques in this section, you must first obtain a reference to ViewFrame's base view, so that you can send messages to it. This can be done with the following statement:

```
local VF := GetRoot().|ViewFrame:JRH|;
```

All the examples in this section assume that the variable `VF` has been set up this way (although you can change it to any variable name you want). It is very important that the reference is stored in a local variable, or used immediately without storing it anywhere. If you put the reference in a slot, and ViewFrame is later removed (perhaps by ejecting the storage card containing it), the slot becomes an invalid reference, and causes an error whenever it is accessed.

### DISPLAYING OBJECTS IN VIEWFRAME

There are several methods provided for displaying objects in ViewFrame:

`VF:?NEWVAL(EXPR);`

This method compiles the string *expr* and executes it. The result becomes the new current object (the previous object path is discarded). This works the same as entering the expression into ViewFrame, except that the special starting characters aren't supported here. ViewFrame is opened if it is closed. Note the use of the conditional message send operator `?:`, which prevents an error if `VF` is `NIL` because it is not installed.

`VF:?NEWVALUE(TEXT, VALUE);`

Makes the object *value* the new current object, using *text* as its description in the object path. Both of these methods return after the new current object is set up, allowing your program to continue executing behind ViewFrame.

Some additional object displaying methods, mainly used by Additions, are detailed in the Additions documentation.

### *VIEWFRAME'S CLOSE CALLBACK*

If your program opens `ViewFrame` and would like to be informed when `ViewFrame` closes, it can specify a callback message that is sent shortly after `ViewFrame` closes. This is done by setting some slots in `ViewFrame`'s base view:

<code>VF.closeContext</code>	the frame to which the message is sent, typically one of your program's views.
<code>VF.closeMethod</code>	the name of the callback message, as a quoted symbol.
<code>VF.closeParams</code>	(optional) an array of parameters to be sent with the message. If this slot isn't set, no parameters are sent.

All of these slots are removed when `ViewFrame` closes, so you need to set them again if you display something else in `ViewFrame`. The slots remain indefinitely if `ViewFrame` isn't closed. You might want to remove them yourself in your base view's `viewQuitScript` so that you aren't surprised by receiving the callback message when your program isn't open.

### *VIEWFRAME'S CURRENT DISPLAY*

If you would like to know what object `ViewFrame` is currently displaying, check the following slots in its base view:

<code>VF.ValPath</code>	An array containing all objects along the object path. The lowest-numbered elements are the oldest. The expression " <code>VF.ValPath[length(VF.ValPath) - 1]</code> " returns the current object. Note that there isn't always a current object – make sure the length of the array is greater than zero.
<code>VF.TextPath</code>	An array containing the text descriptions of the objects on the object path. This always has the same number of elements as <code>ValPath</code> . The expression " <code>stringer(VF.TextPath)</code> " gives you the complete path to the current object.

Note that these slots are not necessarily valid when the close callback message is sent, or at any other time when `ViewFrame` isn't open – garbage collection may have wiped out part of the object path.

Do not add or remove elements in these arrays. `ViewFrame` maintains other arrays parallel to these (such as the current scroll position and display format for each object in the path), and becomes confused if the arrays get out of sync.

## OPENING THE PROGRAMMER'S KEYBOARD

If you would like to be able to open the Programmer's Keyboard by tapping a button in your program, give it this `ButtonClickScript`:

```
func() begin
  local kbd :=
    if GetRoot().|ProgKeyboard:JRH| exists
    then GetRoot().|ProgKeyboard:JRH| ;
    else GetRoot().alphaKeyboard;
  kbd.Toggle();
end
```

Note that this handles the case of the Keyboard not being installed by bringing up the standard on-screen keyboard instead. If you would like to distribute the Keyboard with your program, contact us for licensing details.

## CUSTOMIZING VIEWFRAME

If you do not find the ViewFrame Editor or the Programmer's Keyboard useful, or don't need the ability to open the Toolkit App directly from ViewFrame, you can easily override their buttons in ViewFrame to open any application you want, perhaps another debugging utility or a different on-screen keyboard. This is done by storing the `appSym` of the desired application into specific slots in ViewFrame's base view:

- `editorSym`, to override the Edit button;
- `keyboardSym`, to override the Keyboard button; or
- `toolboxSym`, to override the Toolbox button.

Note that the Toolbox button operates a bit differently than the others – it forces the base view of the opened application to be floating, by modifying its `viewFlags` slot. This is to support early versions of the Toolkit App that aren't floating, preventing it from appearing behind ViewFrame when opened. Not all applications work properly if you make them floating.

The change stays in effect until the machine is reset. To make the change persistent, do it in the `InstallScript` of one of your programs. You have to use a deferred action since ViewFrame's base view may not yet exist when your `InstallScript` is called.

The `DelayTime` slot in ViewFrame's base view was previously documented for adjusting the time that the Delay button would wait before reopening ViewFrame. Since the Delay button has been removed, changes to this slot have no effect.

## WRITING ADD-ONS FOR VIEWFRAME

The ViewFrame Additions mechanism allows a variety of new features to be added to ViewFrame, taking advantage of its existing object browsing features. Possibilities include:

- Commands added to the Additions pop-up menu.
- Viewers that completely replace the built-in display for a particular type of object.
- Annotations added to the usual display of a particular type of object.
- New items added to the object description pop-up.

All documentation relating to Additions is supplied on disk. See “Additions.doc” for details on the Additions supplied with ViewFrame, and how to write your own. Look in the “VF+Sample” folder for a complete NTK project for a sample Addition that you can use as an example and as a base for writing your own.

The add-on mechanism described in the original ViewFrame manual, based on appending entries to ViewFrame’s routing frame, is unsupported and nonfunctional.

## NEWTONSCRIPT SECRETS

*WARNING: The information in this section is provided for its amusement value only. It may be of use in a debugging tool, but don't rely on it in a commercial program (unless, of course, you want people to get mad at you when your program fails on future Newton versions).*

Much of the information previously located in this section, such as internal details of NewtonScript bytecode functions, has been removed. The Newton Formats document, now available from Apple, contains the official details on these and other previously undocumented internal Newton features. Signing a license agreement is required to receive this document, but it does not limit your ability to make use of the information in a debugging or development tool – basically, you just agree not to redistribute the information itself, and acknowledge that Apple may change any Newton internal details at any time without notice (possibly breaking any program based on the Newton Formats information). If you're interested, contact Apple's Software Licensing department (SW.LICENSE@applelink.apple.com, or 512.919.2645) and ask for the Newton Formats license agreement.

### REFERENCES

All NewtonScript objects are represented by a 32-bit reference value. Immediate values are contained entirely within their reference. Other object type references contain a pointer to their actual data which is located elsewhere in memory. The different pointer-referenced object classes (frames, arrays, and binary objects) are distinguished by flag bits in the memory object, not in their reference values. The class of immediate values is encoded in their reference.

The reference value of any object can be turned into an integer with the `RefOf()` global function; such an integer can be turned back into an actual reference with the `Ref()` function. Please note that these functions are not reliable – since NewtonScript integers are only 30 bits long, only a quarter of all possible references have integer equivalents. The Inspector can display full 32-bit reference values since its print function isn't written in NewtonScript, but you may not be able to revisit an object via `Ref()` if any of the highest bits of its reference value are set.

Actually, there was an undocumented NewtonScript syntax that allowed specifying an arbitrary 32-bit reference value: a “#” (pound sign) followed by a 1..8 digit hex number. You may notice that this is exactly the format in which the Inspector displays references. Unfortunately, you cannot enter references this way in the Inspector. Inspector expressions are compiled on the host computer, not the Newton device. Using the #



notation to create a pointer reference would actually create a reference to some memory location in the host computer, not in the Newton device. Trying to transfer the compiled expression to the Newton device for execution would probably crash the host computer, due to the invalid memory reference. All recent versions of the NTK have completely disallowed the # notation, presumably to avoid this problem.

You can freely use the # notation in ViewFrame (perhaps to view an object that was printed to the Inspector) on Newton 1.x devices. However there is no guarantee this will actually work for pointer references, since their reference value can change (even if the object itself hasn't changed, it may have been moved due to garbage collection). Unfortunately, this ability is gone in Newton 2.0.

Different types of references can be distinguished by the lowest few bits of the reference value, as follows:

xxxx0	Immediate values
xxxx1	Pointer references
xxx00	Integers (reference = 4 * value)
xxx01	Normal pointer references (reference = address + 1)
xxx10	Other immediate values
xxx11	Magic pointers: @nnn (reference = 4 * nnn + 3)
x0010	class: Weird_Immediate (ref(NIL) = 2)
x0110	class: Char (reference = 0x10 * Unicode value + 6)
x1010	class: Boolean (ref(TRUE) = 0x1A)
x1110	class: Mark in Newton 1.x, Weird_Immediate in 2.0 (never used)

## TRIVIA

### BOOLEANS

An interesting thing to note: NIL is not a Boolean, and TRUE is not the lowest possible Boolean value. Perhaps NIL was originally intended to have a reference value of 0x0A, but was changed later (because #0A takes three bytes to represent in NewtonScript byte codes, but #02 takes only one byte). Another possible explanation: SELF, the language NewtonScript is derived from, has a false value that is distinct from NIL. Perhaps NewtonScript originally worked the same way, with #0A as the reference value of false.

## CONSTANT-IMMEDIATE VALUES

There are three different forms in which constant immediate values can be represented in a NewtonScript function. A one-byte form handles the values #00 through #06. These include NIL, 0, 1, the null character \$  , and the magic pointer @0. The other two possibilities represent unusable pointer references (even if the referenced addresses were valid, objects located there wouldn't be referenced by a constant). Other constants with reference values that fit in 16 bits are represented with a three-byte form - this handles integers from -8192 to 8191, characters up to \$      , magic pointers up to @4095, and TRUE. Constants outside of that range are placed in the function's literals array, and referenced with a one or three-byte opcode, for a total of five or seven bytes.

In extremely space-critical code, this information can be used to achieve a slight decrease in function size. For example, you can use 1 as a flag value rather than TRUE, saving two bytes per occurrence. If you need an arbitrarily large integer, note that 7777 takes less space to represent in a function than 9999 does.

## STRING CONCATENATION

The string concatenation operators are actually implemented by building an array out of the items, then using `stringer()` on them. The expression

```
A & B && C
```

actually compiles to the following code:

```
stringer( [A, B, " ", C] )
```

Note that this information was given in slightly incorrect form in previous versions of this manual – the space implied by the `&&` operator is implemented with a one-character string, not a character constant. Don't use the `&&` operator if the item to either side of it is a string constant: it's usually more efficient to use `&`, and put an extra space in the string constant.

## TECHNICAL SUPPORT

ViewFrame technical support is provided by its author, Jason Harper. He can be reached in a variety of ways:

Jason Harper  
2907 Pennsylvania Ave.  
Colorado Springs, CO 80907  
719.575.0721  
719.575.0167 (fax)

CompuServe: 76703,4222  
Internet: 76703.4222@compuserve.com  
eWorld (NewtonMail): JasonH

Inquiries about site licenses, sales, and customer service should be directed to:

Creative Digital, Inc.  
293 Corbett Avenue  
San Francisco, CA 94114  
415.621.4252  
415.621.4922 (fax)

cdi@cdpubs.com  
74774.50@compuserve.com  
<http://www.slip.net/~cdi>

## SOFTWARE LICENSE

PLEASE READ THIS LICENSE CAREFULLY BEFORE USING THE SOFTWARE. BY USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, PROMPTLY RETURN THE PRODUCT TO THE PLACE WHERE YOU OBTAINED IT AND YOUR MONEY WILL BE REFUNDED.

1. License. The application, demonstration, system, and other software accompanying this License, whether on disk, in read only memory, or on any other media (the “Software”), the related documentation and fonts are licensed to you by Jason Harper. You own the media on which the Software and fonts are recorded but Jason Harper and or Jason Harper’s Licensor(s) retain title to the Software, related documentation and fonts. This License allows you to use the Software and fonts on a single Newton Product (which, for purposes of this License, shall mean a product bearing Apple’s Newton logo), and make one copy of the Software and fonts in machine-readable form for backup purposes only. You must reproduce on such copy the Jason Harper copyright notice and any other proprietary legends that were on the original copy of the Software and fonts. You may also transfer all your license rights in the Software and fonts, the backup copy of the Software and fonts, the related documentation and a copy of this License to another party, provided the other party reads and agrees to accept the terms and conditions of this License.

2. Restrictions. The Software contains copyrighted material, trade secrets and other proprietary material and in order to protect them you may not decompile, reverse engineer, disassemble or otherwise reduce the Software to a human-perceivable form. You may not modify, network, rent, lease, load, distribute or create derivative works based upon the Software in whole or in part. You may not electronically transmit the Software from one device to another or over a network.

3. Termination. This License is effective until terminated. You may terminate this License at any time by destroying the Software and related documentation and fonts. This License will terminate immediately without notice from Jason Harper if you fail to comply with any provision of this License. Upon termination you must destroy the Software, related documentation and fonts.

4. Export Law Assurances. You agree and certify that neither the Software nor any other technical data received from Jason Harper, nor the direct product thereof, will be exported outside the United States except as authorized and as permitted by the laws and regulations of the United States. If the Software has been rightfully obtained by you outside of the United States, you agree that you will not reexport the Software nor any other technical data received from Jason Harper, nor the direct product thereof, except as permitted by the laws and regulations of the United States and the laws and regulations of the jurisdiction in which you obtained the Software.

5. Government End Users. If you are acquiring the Software and fonts on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees:

(i) if the Software and fonts are supplied to the Department of Defense (DoD), the Software and fonts are classified as “Commercial Computer Software” and the Government is acquiring only “restricted rights” in the Software, its documentation and fonts as that term is defined in Clause 252.227-7013(c)(1) of the DFARS; and

(ii) if the Software and fonts are supplied to any unit or agency of the United States Government other than DoD, the Government’s rights in the Software, its documentation and fonts will be as

defined in Clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA supplement to the FAR.

6. **Disclaimer of Warranty on Software.** You expressly acknowledge and agree that use of the Software and fonts is at your sole risk. The Software, related documentation and fonts are provided “AS IS” and without warranty of any kind and Jason Harper and Jason Harper’s Licensor(s) (for the purposes of provisions 6 and 7, Jason Harper and Jason Harper’s Licensor(s) shall be collectively referred to as “Jason Harper”) EXPRESSLY DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. JASON HARPER DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE SOFTWARE AND THE FONTS WILL BE CORRECTED. FURTHERMORE, JASON HARPER DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE AND FONTS OR RELATED DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY JASON HARPER OR A JASON HARPER AUTHORIZED REPRESENTATIVE SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY. SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU (AND NOT JASON HARPER OR A JASON HARPER AUTHORIZED REPRESENTATIVE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

7. **Limitation of Liability.** UNDER NO CIRCUMSTANCES INCLUDING NEGLIGENCE, SHALL JASON HARPER BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES THAT RESULT FROM THE USE OR INABILITY TO USE THE SOFTWARE OR RELATED DOCUMENTATION, EVEN IF JASON HARPER OR A JASON HARPER AUTHORIZED REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. In no event shall Jason Harper’s total liability to you for all damages, losses, and causes of action (whether in contract, tort (including negligence) or otherwise) exceed the amount paid by you for the Software and fonts.

8. **Controlling Law and Severability.** This License shall be governed by and construed in accordance with the laws of the United States and the State of California, as applied to agreements entered into and to be performed entirely within California between California residents. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this License shall continue in full force and effect.

9. **Complete Agreement.** This License constitutes the entire agreement between the parties with respect to the use of the Software, related documentation and fonts, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Jason Harper.



---

## MORE CREATIVE DIGITAL NEWTON DEVELOPER TOOLS

### ***PDA DEVELOPERS™***

PDA Developers provides in-depth technical information about developing PDA software, focusing on Newton, Psion, GEOS, Magic Cap, and Hewlett Packard devices. Each issue includes news and announcements, programming tips and techniques, product reviews and previews, and programs with in-depth developer descriptions. For all levels.

### ***PDA DEVELOPERS – SOURCE CODE DISK***

Each issue of the *PDA Developers* source code disk includes the text of each article, the source code for each program in the issue, plus developer-oriented freeware, shareware, and goodies. For subscribers that receive just the disks, we include a Common Ground image of the printed issue. Available in Mac and Windows versions.

### ***GIZMOBEAM™ - MAC/NEWTON IR***

GizmoBeam is a device driver for beaming between Mac programs and Newtons. It includes sample code with Think C 7.0 source and detailed documentation. Requires knowledge of the Macintosh Device Manager and Serial Device Drivers. Requires CE-IR2, 3, or 4 hardware. Distribution licenses extra.

### ***MICROWAVE™ - DOS/NEWTON IR***

MicroWave is a linkable library for beaming between Newtons and DOS-based hardware. It includes a C header file, Microsoft and Borland libraries, three samples with full source, and docs. Requires CE-IR2, 3, or 4 hardware. Distribution licenses extra.

### ***RAGOUT™ (RAGU') - THE ULTIMATE SOUP TOOL***

Designed for Newton developers, consultants, and power users, Ragout lets you:

- Create, delete, copy, and move soups, soup indices, and soup entries
- View, and edit soup entries, and change slot data types at any frame depth;
- Create multiple copies of an entry for sizing soups.
- Move to the first and last entries, forward and backward one or N entries, go to a specific tagged entry, or search for an entry by index value; and
- Fax and print soup information and entry details, and beam soup entries.

***TO ORDER ANY OF THESE PRODUCTS, CONTACT CREATIVE DIGITAL.***

