ViewFrame 1.2 Additions

Introduction

Starting with version 1.1, ViewFrame supports a mechanism for externally adding new features, known as ViewFrame Additions. The add-on mechanism described in the version 1.0 manual is now obsolete. Additions are contained in separate packages which may be installed or removed at any time (even while ViewFrame is open), and may be installed on a different store than ViewFrame itself. Many of ViewFrame's features are supplied in a series of Addition packages, which are the subject of this document. Splitting up the whole program into multiple packages offers you, the user, several benefits:

- Features that you don't want aren't taking up any of your precious storage space. Install only the features you're interested in, remove them if you don't find them useful.
- You can have multiple configurations of ViewFrame depending on the work you're doing. For example, you could have ViewFrame by itself in internal memory for debugging a program that uses a modem or other special card (preventing the use of a storage card), and put the Additions that you use when working on other programs on a storage card.
- Additions make it easier to distribute new features and bug fixes (assuming that the feature or problem can be addressed by an Addition, rather than requiring a change in ViewFrame itself). Since new or changed Additions are of no use to anyone but ViewFrame owners, they can be distributed via public online services. Additions are expected to be added and improved on a fairly frequent basis, which is why they are documented in this file rather than in the printed ViewFrame manual.
- If you have a need for some custom object viewing or manipulation ability, you can write your own Addition to perform it. This allows you to take full advantage of ViewFrame's existing object browsing and display features, while easily adding on your own features. A sample NTK project is supplied on disk that you can use as a shell for writing Additions of your own.

Additions also make life easier in several ways for me, the author, which indirectly benefit you by making it possible for me to implement more features in the same amount of time:

- I can develop and test Additions much faster than changes to ViewFrame itself: they're smaller, so they compile faster, and I can leave ViewFrame open (keeping the object I'm testing the Addition on as the current object) while downloading changes to an Addition.
- I no longer feel constrained to add only features that have a high benefits/code size ratio. If a particular feature would take a lot of code to implement, or would only be of use to a small percentage of Newton developers, I now feel free to implement it anyway, knowing that I'm not forcing anyone to waste storage space on the feature who doesn't want it.

Items marked (NEW) are being introduced for the first time with this version (1.2) of ViewFrame.

Using Additions

 The Additions button in ViewFrame, used to invoke Addition commands

Additions are supplied as individual packages. These are installed and removed just like any other Newton software, except that they do not appear in your Extras Drawer. Additions can be installed and removed at any time, even if ViewFrame is open. Newly installed Additions are available for use immediately, with no additional action required on your part. There may be a problem if you remove an Addition that produced the current object, or some prior object on the path, but the worst that should occur is an error message. If this happens, enter an expression or use an Addition that creates a new object path, or back up to an object prior to any that that were produced by the removed Addition package.

There are several basic kinds of Additions:

Viewers

Viewers automatically give you an improved object display whenever an object of appropriate type is selected. This entirely replaces the normal display of the object. No user action is required to use a viewer, other than selecting an object of the type that the viewer expects. If you'd like to see an object in its raw form, rather than as interpreted by a viewer (either built-in or an Addition), tap the Fmt button and select the Alternate format.

Annotations

Annotations add some additional data, which may either come from a built-in viewer or an Addition, to the top of the object display. The data may just be additional labeling for the object, or it may be an item that can be tapped on for further information.

Commands

Commands are features that must be specifically invoked by the user. They may send information to the Inspector, display information in a separate view, add to or replace the current object path, or almost anything else. Tapping on the Additions button pops up a list of available commands. Only items relevant to the current object, or items that don't require a current object, are shown in the popup menu at any given time. There is a dividing line between Additions from separate packages. Most packages contain an item with a name starting with "About", that gives a copyright notice and a list of all functions included in the package. To save space, all such items are collected into a separate list, accessed by an "About..." item at the bottom of the main commands list.

Info Pop-up Items

Pop-up items add additional data to the pop-up accessed by tapping on the object description line (just above the object display area). Typically, these items show attributes of the current object that aren't commonly needed, or that take a significant amount of time to generate, making them inappropriate as an annotation or other automatic information display. Currently, the only example of this type of Addition is the Image Of View, in the VF+Dante package.

Commands and viewers can work together to provide more complex features. An example is the Find Graphics command in the VF+Transfer package. This command builds an array of the `icon` slots found in all currently open picture views, and makes it the new current object. A viewer in the same package then recognizes this array and displays it in a special format (showing all of the pictures in graphical form at the same time, rather than requiring that they be individually selected from the array, as would be the case if ViewFrame displayed it normally).

If a command produces a new current object, it also adds a short textual description of the action it performed to the object path. This allows you to tell, by looking at the object path, what

steps you took to reach the current object. The added text generally doesn't conform to NewtonScript syntax, as do objects added to the path by tapping on them – if a command's action was expressible by a short NewtonScript code fragment, it probably wouldn't have been implemented as a command in the first place. I would have been an expression added to the expression pop-up list.

VF+General

Version documented: 1.2

This package adds various general-purpose object manipulation features to ViewFrame. A lot of the commands relate to soups, and may be split off into a separate VF+Soup package at some point.

Get App

Available: Always, on Newton 1.x systems only.
Object: Replaced with new object.

Allows selection of any installed application. The partFrame of the app (as found in the `GetGlobals().extras array`) becomes the new current object. You can get to the app's base view by tapping the `formContext` slot, and to its base template by tapping `theForm`.

Newton 2.0 does not support the `extras` array, so this command is not available. See the Get Part and Get Base View commands in the VF+Dante package for equivalents that work under Newton 2.0.

(App Icon)

Available: After an app has been selected using the Get App command.
Object path: Replaced with new object.

Reselects the `partFrame` of the app most recently selected using Get App. It is located in the `GetGlobals().extras` array by the name of the app. Installation or removal of apps won't affect this feature, as long as the selected app hasn't been renamed, or removed without being replaced. To remove the app icon from the Additions popup list, select Get App again and tap outside of the list of apps to dismiss it.

Find Methods

Available: When the current object is a frame containing executable code, or has `_parent` or `_proto` pointers which presumably lead to such code.
Object path: Not directly affected.

Gives a pop-up menu listing all methods defined in the current frame, or reachable from it using inheritance. The pop-up can be scrolled by tapping the arrows at the top and/or bottom. Selecting a method generates an expression to call it, in ViewFrame's expression entry area. If the method takes parameters, a view is first brought up to allow you to specify values for the parameters. To actually perform the method, tap the Eval button.

As it first appears, the list of methods is in the normal inheritance search order (the frame itself first, then its proto, then its parent). The path from the current object to the location of each method is shown by a pattern of >'s (indicating `_proto` links) and ^'s (indicating `_parent` links). For example, "^^>" indicates that the following methods are found in the prototype of the grandparent of the current frame. If the root view (as returned by `GetRoot()`) is present in the path, "(root)" is shown in the corresponding place in the path indication. Methods listed before any path indication are actually located in the current frame. Methods that are overridden by a method of the same name, closer to the current object in the inheritance path, are shown enclosed in parentheses and are not selectable.

The popup list starts with a Sort By Name item: selecting this alphabetizes the list. Path indications and overridden methods are dropped from the list.

WARNING: This Addition makes ViewFrame somewhat dangerous to use, since it is now possible to perform destructive operations merely by tapping. Do not call any method unless you understand what it does.

Slot Symbols

Available: If the current object is a frame.
Object path: New object added.

Generates an array containing the slot names (as opposed to the slot values, which is what are normally of interest) for each slot in the current frame. This makes it possible to examine the slot symbols, perhaps to see if any are contained in a package and might therefore cause problems after that package is removed. This is only useful on Newton 1.x systems: under Newton 2.0, the Location format accessible from the Fmt button gives you a lot more information about the slot symbols in a frame.

(NEW) Get Store

Available: Always.
Object path: Replaced with new object.

Pops up a list containing the names of all available stores. Selecting one makes that store the new current object. If there are any package stores installed, they appear in the list after a dividing line.

Store Info

Available: If the current object is a store.
Object path: New object added.

Gives you a frame containing the results of various information-getting methods defined for stores: `GetName`, `GetKind`, `GetAllInfo`, `GetSignature`, `GetSoupNames`, `IsReadOnly`, `IsValid`, `TotalSize`, `UsedSize`, `Overhead`, `GetPackageDirectory`, and `HasPassword`. The name of each slot is the name of the method that provided the info for that slot. Slots are only present if the store implements the corresponding method. Some of the listed methods are only found under certain system versions. Some are undocumented and may go away. Also present is a `percentFull` slot, calculated from the total and used size.

Get Soup

Available: If the current object is a store.
Object path: New object added.

Allows you to select any soup on the store represented by the current object.

Get Union Soup

Available: Always.
Object path: Replaced with new object.

Allows you to select a soup name, and gives you the union soup object of that name. The names shown are the names of all internal soups. Not all of them exist on any other stores, but you can still access them as union soups. This Addition doesn't do anything you can't already do with the "GetUnionSoup(" ")" command in ViewFrame's expression pop-up, but is easier to use since you don't have to type in a soup name.

Soup Info

Available: If the current object is a soup.
Object path: New object added.

Gives you a frame containing the results of all various information-getting methods for soups, much as the Store Info Addition works for stores. For individual (non-union) soups, the slots may include: `GetName`, `GetAllInfo`, `GetInfoModTime`, `GetIndexes`, `GetIndexesModTime`, `IndexSizes`, `GetNextUID`, `GetSignature`, `GetSize`, `GetStore`, `GetFlags`, `HasTags`, `GetTags`, and `IsValid`. As with Store Info, slots are only present if the soup object implements the corresponding method, some of which are system-specific or undocumented. Also present is `simpleCursor`, the result of performing the simplest possible query (`Query(`*`soup`*`, {type: 'index})`) on the soup.

For union soups, most of the information methods listed above are invalid. A shorter list of methods is tried: `GetName`, `GetSize`, `HasTags`, `GetTags`, and `IsValid`. The `simpleCursor` slot is present as before, plus slots named `0`, `1`, and so on, containing the soup `info` frames for each component soup of the union soup. If the soup isn't present on a particular store, its slot is `nil`.

Cursor Info

Available: If the current object is a cursor.
Object path: New object added.

Gives you a frame containing information extracted from the cursor. This works the same for cursors on either individual or union soups. The data items are:

`entry` The current entry that the cursor is pointing to.
`numEntries` The count of undeleted entries that the cursor can access.
`currentPos` The position of the cursor with its accessible entries. This is nil if the cursor is positioned beyond either end of its entries, and is meaningless if the cursor is pointing to a deleted entry.
`dirtyEntries` The count of entries for which `FrameDirty()` returns true.
`deletedEntries` The count of entries which return `'deleted`.
`totalSize`
`totalTextSize` The grand totals of applying the `EntrySize()` and `EntryTextSize()` calls, respectively, to each entry that the cursor can access.

Some additional slots may be present under Newton 2.0, giving the results of applying various information-getting methods to the cursor: `CountEntries`, `status`, `WhichEnd`, and `indexPath`.

Object Size

Available: If the current object is anything other than an immediate value.
Object path: New object added.

This is a partially successful attempt at determining the amount of memory a given object occupies. It works by trying each of the four cloning functions (`clone`, `DeepClone`, `TotalClone`, and `EnsureInternal`) on the object, noting the amount of temporary decrease in free memory due to the existence of the copy. The results are returned in a frame, containing slots with the name of each cloning function, with a value equal to the corresponding change in free memory. There is also a `freeMem` slot, containing the total amount of free memory at the time this Addition was used. If any of the functions runs out of memory, its slot will have a `nil` value: the actual value can be assumed to be something larger than the `freeMem` value shown. Note that this function may take a few minutes to execute, if the object is too big to make

an entire copy of.

The `clone` slot gives you the size of the object's top level. The `DeepClone` slot should give you the total size of the data belonging to the object, but not the size of frame maps and slot/class symbols that may be shared with other objects. `TotalClone` should give you the actual total size of the object, excluding any portions that are in ROM. `EnsureInternal` can give any result from 0 on up to the `TotalClone` value, depending on how much of the object is in RAM already.

In practice, this function returns inconsistent results, of uncertain interpretation. In particular, the `TotalClone` result is occasionally negative, indicating that copying the object actually increased the free memory in the Newton! The results should therefore be considered as rough approximations to the object size at best.

(NEW) Under Newton 2.0, the returned frame may also contain a `TrueSize` slot, containing the result of applying the `TrueSize()` debugging function to the object. This result is a frame containing breakdowns of the total size by object class. The `objects` slot in this frame contain the overall total size. Note that `TrueSize` only counts objects in the frames heap – it never follows references to read-only objects.

Make Picture

Available: If the current object is anything other than an immediate value.
Object path: New object added.

Attempts to convert an object into a picture, using the `MakePict` global function. If this fails, `MakeShape` is tried, with `MakePict` applied to its result. The complete set of object types that these functions work on is not documented, so this Addition can be tried on any non-immediate object. If no picture can be generated, the Newton beeps.

This produces a picture out of the results of any of the shape creation calls, arrays of shapes, bounds frames, icons, polygonShape objects, and perhaps others. Upon any successful conversion, the resulting picture is displayed by ViewFrame's normal picture display capability. The picture is created using a custom dark pattern for lines, and a custom light pattern for fills, so you can tell whether the original object specified its own line/fill patterns.

Under Newton 2.0, this command work on views, giving you a picture shape that encapsulates the current appearance of the view. This is an interesting alternative to Newton screen shots, such as performed by the NTK, since the results are object-based: the picture prints at full printer resolution, not 72 DPI as a normal bitmapped screen shot would. However, the process often fails if applied to the root view, due to lack of memory.

Bookmark >Inspector

Available: If there is a current object.
Object path: Not directly affected, but using a bookmark replaces the object path with a new object.

Prints a bookmark to the NTK Inspector, which can be used later to return to the current object. The bookmark is used by placing the insertion point anywhere within it, and hitting Enter. Each bookmark starts with an empty comment, which you can fill in to remind you of what the original object was. You may also want to edit the "bookmark" string inside the bookmark, to customize its appearance in the object path.

Bookmarks work by generating a NewtonScript expression that duplicates the current object path. A requirement for them to work properly is that every element of the object path must be generated by a valid NewtonScript operation, including ViewFrame's extended expression syntax that starts with a slash. Almost any object path that is generated entirely by expressions entered in ViewFrame (perhaps with the aid of the expression popup menu), or by tapping on slots in frames and arrays, meets this requirement. There is one possible exception: slot names containing non-alphanumeric characters (such as app symbols, which contain a

colon) may have to be manually enclosed in vertical bars for the bookmark to work.

Objects added to the path by Additions may not be expressed in NewtonScript, and any bookmark made with such an object in the path generates an error if you try to use it. Generally, such objects appear in the object path enclosed in < and >, or prefixed by ->. As a special case, object paths that start with a Get App command (or by selecting the app icon remembered from the previous Get App) produce a valid bookmark, even though Get App doesn't express its action in NewtonScript. Future versions of this Addition may allow additional types of Addition-produced objects to be included in bookmarks.

Bookmarks exactly repeat the steps originally taken to get to the current object, and have no ability to respond to changes in the environment. For example, if you are looking at the base view of a particular app on a Newton 1.x system, the object path might look like:

```
GetGlobals().extras[17].formContext
```

A bookmark made from this path will always return to element 17 of the `extras` array. If the app has been replaced, or another app installed or removed, element 17 could refer to a different app. In this case, the Get App Addition would be a more reliable way to return to the same app, since it remembers apps by their names.

If the original object path contains any operations that created objects, then a bookmark made from that path recreates those objects from scratch, rather than refering to the original objects. It cannot possibly do so, since the original objects could have been long since reclaimed by the garbage collector by the time the bookmark is used. This may occasionally affect the usability of bookmarks. As an example, consider an object path created by getting a soup, then performing a query on it to get a cursor. A bookmark made from this path generates a brand new cursor, which may not share all characteristics of the original: in particular, it may not point to the same entry.

(NEW) X := Object;
Available: If there is a non-NIL current object.
Object path: Not directly affected, but executing the Inspector expression to revisit the object add its new value to the object path

Stores a reference to the current object in the global variable X. This allows you to easily access it from the Inspector, perhaps to perform some operation on it that wouldn't be convenient to do in ViewFrame's expression entry area. Also, the following expression is written to the Inspector window:

```
GetRoot().|ViewFrame:JRH|:AppendValue("/Revisited",X); X:=nil
```

Moving the cursor back to that line and pressing Enter causes the variable X, with any modifications you've done to it, to be redisplayed in ViewFrame. Also, X is set to `NIL`, to allow the object to be garbage collected later, and to prevent any errors that might occur if the object references a package which is later removed. That part of the expression can be deleted before pressing Enter if you don't want that feature.

VF+Transfer

Version documented: 1.0

This package adds file transfer capabilities to ViewFrame. Transfers can be made to any computer with a standard XModem protocol implementation. The required terminal settings are: 9600 baud, 8 data bits, no parity, and 1 stop bit. Hardware (or no) flow control should be used: software flow control locks up as soon as an XOFF character is encountered in the data being sent. This Addition uses the absolute basic XModem protocol: no CRC, no 1K blocks. Most implementations fall back to these settings, but you may save a minute or two per transfer by

telling your communications software to use these settings as the default.

Depending on the data being sent, you may be given an Add MacBinary Header option. MacBinary is a standard protocol for including Macintosh-specific file information inside a file, so that it can survive transfer through non-Macintosh systems. If data is being transferred to a Macintosh, you will almost always want to leave this option enabled, so that the file is properly identified on the Macintosh. Otherwise, you will get the dreaded ”The application that created this document cannot be found” message.

When transferring to a non-Macintosh system, you generally want to disable MacBinary. If enabled, you must use a MacBinary decoder on the receiving machine. This is extra work, but it does have the advantage of preserving the exact length of the data, which plain XModem cannot do. A non-MacBinary transfer pads the data with nulls to an exact multiple of 128 bytes.

Find Graphics

Available: Always.
Object path: Replaced with new object.

This feature has no inherent connection with file transfers, but it is very handy for getting references to graphic objects to be transferred, so it got stuck in this package. It finds every open view (including those completely obscured by other views) that contains an `icon` slot (which can contain either a bitmap or picture object), and builds an array of all of these objects. Views belonging to ViewFrame are skipped over. A special viewer then shows all of the graphics in this array at once, rather than having to select each element individually in order to see it. Tapping on one of the item number lines selects that graphic, just like tapping an element in a normal array listing.

Suggestion: Close the Extras Drawer before using this function, unless it’s one of the Extras Drawer icons that you’re interested in finding.

Binary Object >XModem

Available: If the current object is a binary object.
Object path: Unchanged.

Transfers an arbitrary binary object to another computer. If MacBinary is used with this command, the file is transferred with a type of “DATA” and a creator of “????”. The file won’t be openable on a Macintosh by double-clicking it. Sending it with MacBinary, however, may still be useful since it preserves the exact length of the object.

Picture->PICT >XModem

Available: If the current object is a frame or binary object of class 'picture.
Object path: Unchanged.

Translates a Newton picture object to a Macintosh picture (which is simply a matter of adding 512 null bytes at the start), and transfers it to another computer. If transferred to a Macintosh with MacBinary enabled, the graphic is recognized by all Macintosh applications as a PICT file, and can be opened with TeachText or SimpleText if double-clicked. Support for this format on non-Macintosh computers is minimal. Any text objects in the picture are lost, since they are in a different format in Newton pictures (required to support Unicode characters).

Bitmap->MacPaint >XModem

Available: If the current object is a bitmap (a frame containing slots named `bounds`, `bits`, and (optionally) `mask`), or a binary object of class `'bits` or `'mask`.
Object path: unchanged.

Translates a Newton bitmap object to the simple black and white graphics format used by early versions of MacPaint, and usable by many viewers and graphics conversion programs on various computers. Note that the MacPaint format has a fixed size of 576 by 720 pixels, so there is generally a lot of white space in the resulting file. Transferring to a Macintosh with MacBinary enabled produces a file that tries to open GIFConverter (a popular shareware graphics conversion utility) when double-clicked. If you would prefer to transfer a Newton bitmap to a Macintosh PICT file, use the Make Picture command in the VF+General Addition, then use the picture transfer function mentioned above.

VF+Binary

Version documented: 1.2

This package adds various features for analyzing and displaying binary objects. Many of these Additions send output to the Inspector rather than display it onscreen, due to the volume of data generated.

Histogram

Available: If the current object is a binary object, at least 4 bytes long.
Object path: Unchanged.

Displays a bar graph showing the frequency of occurrence of every possible byte value in the current object. The graph is oriented vertically, with byte value 0 at the top and value 255 at the bottom. Tapping on the graph, or tapping the scroll arrows to scroll through the graph, hilites a bar with triangles at each end, and displays information about the byte value represented by that bar in the area below the graph. This information includes the byte value (in decimal, hexadecimal, and character form), the number of bytes in the object which have that value, and the percentage of the object size that this number represents.

The histogram initially shows you frequency data based on every byte in the object. It can also show data based on subsets of the object: either every other byte (even bytes only or odd bytes only), or every fourth byte (starting at 0, 1, 2, or 3 bytes into the object). Comparing the histograms of these subsets can reveal additional information about the structure of the object. For example, if the even bytes only and odd bytes only histograms are substantially different, then you know that the object contains data organized into 16 bit (or multiple thereof) elements.

(NEW) The routine that analyzes the binary object is now native compiled, making this command much, much faster.

Some objects to try this command on:

`@320` a picture.
`@102` `samples`, a sound.
`@145` an object named `ROM_ParaGraphCodeBook2`. The exact purpose of this object is unknown, but it's a good example of the use of the data subset options: all of them produce distinctly different results, indicating that the object's structure is based on elements of at least four bytes.

Dict >Inspector

Available: If the current object is a binary object that appears to be an enumerated dictionary (the test used is not conclusive). Typically such objects are of class `'dictdata` or `'AirusA`, and are often found in `GetGlobals().dictionaries` in Newton 1.x systems. Unfortunately, there doesn't seem to be an easy way to access the built-in Newton 2.0 dictionaries in a form that this command can recognize.
Object path: Unchanged.

Sends a list of the words contained in the dictionary to the Inspector. When it's done, the total number of words found is displayed in ViewFrame where the object path normally appears. This may be useful even if the Inspector isn't connected.

Some dictionaries contain an attribute byte for each word. If this is present, the attribute byte is displayed in parentheses after each word for which the byte is not zero. In many dictionaries, a value of 128 indicates a word that should be capitalized, and 192 indicates a word that should appear in all caps. The meaning of other attribute values is not documented.

This command does not currently support lexical dictionaries such as the phone numbers dictionary, which specify a pattern for words to match, rather than an explicit list of words.

Picture >Inspector

Available: If the current object is a binary object of class `'picture` or `'picturedata`.
Object path: Unchanged.

Sends a listing of the opcodes contained in a picture object (either Macintosh or Newton generated) to the Inspector. Not all opcodes are currently supported, such as the Macintosh color-related PICT opcodes that are meaningless on the Newton. If any opcode is listed as being unknown or unimplemented, do not trust any further output produced by this command, since it may have gotten out of sync with the picture data.

For a complete discussion of PICT opcodes as used on the Macintosh (which include all of the Newton opcodes except for the Unicode text extensions), see Appendix A of Inside Macintosh: Imaging with QuickDraw.

DisARM >Inspector

Available: If the current object is a binary object of class `'code`, or is a frame containing such an object in a slot named `code`.
Object path: Unchanged.

Disassembles ARM610 code. Starts at the beginning of the binary object, or, if a frame is the current object, uses the contents of the `offset` slot in the frame to tell where to start. In either case, disassembly continues through the end of the code object. This feature works on certain system patch routines and library routines, and native compiled functions in packages. It cannot do anything with the built-in native functions in the Newton, since their code exists outside of the NewtonScript environment. If the length of the code object isn't a multiple of four bytes (the size of an ARM instruction), the leftover bytes at the end are displayed with the label "Extra byte:".

Explanation of the ARM opcodes that are displayed is beyond the scope of this documentation. A good reference is the book The ARM RISC Chip, by Alex van Someren and Carol Atack (Addison-Wesley, ISBN 0-201-62410-9). Even with the book, the results may not make any sense: RISC assembly language is not known for being highly human-readable.

(NEW) Font >Inspector

Available: If the current object is a binary object of class `'sfnt`.
Object path: Unchanged.

Dumps selected information about a TrueType font to the Inspector. Suitable objects can be found within the items of `GetGlobals().fonts`. The offset and size of all tables within the font are shown, but in most cases no further interpretation is attempted. More information is given for the following tables:

bloc	the point sizes of bitmaps contained in the font are shown.
cmap	the ranges of characters contained in the font are shown.
name	all name strings from the font are shown. Interestingly, every Newton font,

whether built-in or converted, appears to be named “Apple Roman Regular”. The Newton system software apparently does not use this aspect of TrueType.

Viewer: Polygon Shapes

Used: When the current object is a binary object of class `'polygonShape` (typically found in the `points` slot of a `clPolygonView`, or returned by the `ArrayToPoints()` global function).

Displays the type, number of points, and point coordinates of a polygon shape. Also displays a graphical representation of the shape.

VF+Function

Version documented: 1.1

Viewer: Bytecode (Interpreted) Functions

Used: When the current object is a bytecode function object, identified as a frame containing slots named `instructions`, `literals`, and `numArgs`. These objects have class `'CodeBlock` if intended to run on all Newton devices, or class `'_function` (actually, a `Weird_Immediate` value that is automatically translated into the `'_function` symbol) if intended for Newton 2.0 only.

Displays a reconstruction of the NewtonScript source code that generated this function. General details on this feature are contained in the ViewFrame manual, in the “Function Listings” section of the “Interpreting Results” chapter.

This feature does not work directly on native compiled functions. However, such functions normally contain a bytecode version of the function in a slot named `bcFunc`, for the benefit of any future Newton models that are based on a different processor, and therefore cannot execute the ARM processor native code. Tapping on the `bcFunc` slot gives you a listing of the bytecode version of the function. If the slot doesn't exist the bytecode version was suppressed, and VF+Function cannot display the function. You can still examine the function in disassembled form, using the DisARM command in the VF+Binary Addition package, but the results are not likely to be very meaningful.

(NEW) VF+Function now handles the new, more efficient form of functions that is available for Newton 2.0-only packages. The new form does not normally contain the names of parameters and local variables, although it may contain a `DebuggerInfo` slot giving this information. VF+Function extracts the names from the `DebuggerInfo` slot if present, or it assigns arbitrary names of the form `Arg`*n* and `Local`*n* to the parameters and local variables of the function. You may see some local variables with a declaration preceded by the comment “/* closed */”. These are variables that are referenced from functions nested within the current function: such variables require special handling under the new form.

Some known limitations of function listings:

- Extremely large or small floating-point constants may be listed incorrectly, due to limitations on the range in which `SPrintObject()` and other real-to-string conversions work properly.
- If a function uses any reserved NewtonScript keywords as variable or slot names (which is bad form, but can be done if the keyword is enclosed in vertical bars), the listing will not include the vertical bars, and will therefore not be valid NewtonScript code.
- Any changes in the exact format of compiled functions are very likely to break VF+Function. The high accuracy of function listings with VF+Function comes at the cost of flexibility in handling unexpected patterns of bytecodes.

(NEW) Find PC

Available: If the current object is a bytecode function, as described above.

Object path: A clone of the function is added to the object path, with a slot added to indicate the desired PC value. If the current object is already such a clone, it is reused rather than making a new copy of the function.

Opens a slip allowing entry of a PC (Program Counter) value, in the range of 0 through one less than the size of the function's instructions object. The function is then listed again, with flags of the form «*nnn*» surrounding the portions of the listing that correspond to bytecodes near the specified PC value. Note that there may be nothing flagged with the exact PC value you entered – it may lie between two bytecode instructions, or it may correspond to internal compiler-generated code that doesn't appear in the listing. Therefore, all instructions within five bytes of the specified location are flagged, in the hope that they bracket the location of interest even if it does not get flagged itself.

This command is typically used in conjunction with the `StackTrace()` debugging call, which can be used while in a break loop, to show you the chain of function calls that led up to the point of the break. The current position of execution within each function in the chain is shown as a PC value, which isn't normally very meaningful. This command allows you to turn that PC value into a visible position within the function's listing.

Keep in mind that the NewtonScript bytecode language is fundamentally different from NewtonScript itself, making it difficult to determine exactly what part of the NewtonScript code corresponds to a particular bytecode instruction. The bytecode language uses a postfix representation: all operands are fully evaluated before the operation that uses them is even specified. Contrast this to NewtonScript, where almost all operations are written between or before their operands.

To illustrate the ordering differences between NewtonScript and bytecodes, consider the following simple function:

```
func() a := b + c;
```

The compiled form of the function has six bytes of bytecode instructions, so using the Find PC command with a value of three flags every instruction. The results are:

```
func() begin
   «3»a := «2»«0»b«0» + «1»c«1»«2»«3»
end
```

Note that the numbering of instructions is almost completely backwards relative to the order in which they appear in the NewtonScript source. Instructions 0 and 1 retrieve the variables `b` and `c`. Instruction 2 then adds them, and instruction 3 stores the result in the variable `a`. The remaining two bytes of instructions relate to the implied return value of the function: they don't explicitly produce any part of the listing, and therefore no flags appear for them.

In general, to determine what is happening at a particular PC value, look for text between the two corresponding flags that is not also enclosed in earlier-numbered flags. For example, if this function thows an exception and a stack trace shows the offending PC location to be 2, the cause of the exception is the addition being performed. Everything else between the `«2»` markers is also between the `«0»` or `«1»` markers, and would have generated an exception with a PC value of 0 or 1 if they had been the source of the problem.

.c1.VF+Intercept

Version documented: 1.1

NOTE: A planned addition to VF+Intercept for this release was a single-step debugging feature, based on Apple's NS Debug Tools package. The final version of NS Debug Tools is not

yet available at the time of this writing. The single-step debugger is intimately tied to internal details of that package, so its current form will almost certainly fail to work under the released NS Debug Tools. Therefore, the single-step debugger has been temporarily removed from VF+Intercept. As soon as NS Debug Tools are publicly available, and the debugger updated to work with them, a free upgrade to VF+Intercept will be made available via online services.

This package adds the Interceptor to ViewFrame, a feature for catching calls to a specified method and responding by displaying debugging info or allowing the action of the call to be modified. Intercepts can be installed in any RAM-based frame (typically a view) that contains a `_proto` slot leading to a frame (typically a view template in ROM or a package) containing some executable functions. Intercepts will not work in any case where parent inheritance is required to find the method, since the intercept routine uses `inherited:`*`methodName`* (which uses only proto inheritance) to call the intercepted routine. Also, the intercepted method must not be contained directly in the frame where the intercept is installed, since the intercept would overwrite the method in this case. If an intercept is already installed for a particular method, it must be removed before another intercept for the same method can be installed.

Intercepts are simply small compiled NewtonScript routines that perform the specified debugging operation and call the original method. They reside entirely in the frames heap, and do not require the continued presence of ViewFrame or VF+Intercept once installed (this may not be true of all future types of intercepts, such as a step/trace debugger).

The Interceptor does not currently keep track of installed intercepts. This would cause problems whenever a package containing intercepted methods was removed or replaced, since any references to the former intercepts would now contain invalid pointers. For now, you need to keep track of intercepts yourself, so that you can remove them if needed. Resetting the Newton removes all intercepts.

Install Intercept

Available: If the current object is a writable frame, with a `_proto` slot that leads to any executable routines that can be intercepted.

Object path: Updated to reflect the addition of an intercept routine to the current frame.

Brings up a screen where a particular method can be selected, and other details specified for an intercept. Tapping the Install button creates an intercept routine in the current frame.

Remove Intercept

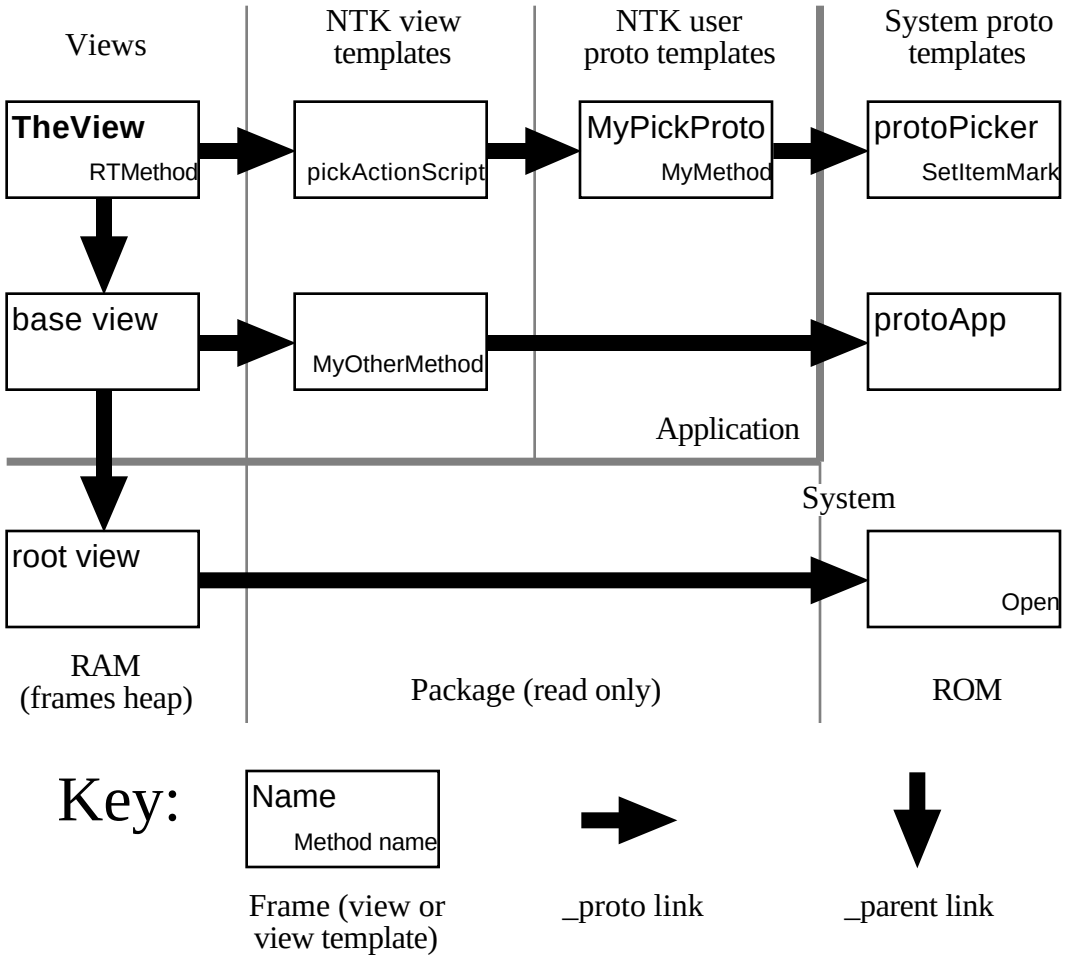
Available: If the current object is a frame containing one or more intercept routines (identified by the presence of an unusual character constant in the routine, unlikely to be found in any normal routine).

Object path: Updated to reflect the removal of an intercept routine from the current frame.

Pops up a list of the intercept routines installed in the current frame. Tapping on one removes the routine from the frame. Currently, you can do the same thing by scrubbing out the slot holding the routine. However, future versions of the Interceptor may maintain internal information about installed intercepts. Use this command, rather than scrub out the slot, to ensure that any internal information stays consistent with the intercepts actually installed.

What Can and Can't be Intercepted

This illustration shows the view hierarchy of a hypothetical application. It is based on the system `protoApp` template, and contains a single child view named `TheView`, which is based on a user proto which is based on the system `protoPicker` template. Some methods of interest, all of which can be legitimately sent to `TheView`, are named in the frames which contain their implementation. However, not all of them can be successfully intercepted.



`TheView:pickActionScript()`
`TheView:MyMethod()`
`TheView:SetItemMark()`
OK. Intercepts for any of these three methods can be installed when `TheView` is the current object. Note that it makes no difference how far down the proto chain the method is located, nor whether the method is called from the program or by the system. Note that `TheView`'s template and proto template are also valid places to intercept a call to `TheView:SetItem-Mark()`, except for the fact that those frames are read-only and cannot accept the addition of an intercept routine.

`TheView:RTMethod()`
Disallowed. Installing an intercept in this situation overwrites the method to be intercepted. Note that this is a fairly uncommon situation: the method is presumably generated at runtime, or selected from a set of existing methods.

`TheView:MyOtherMethod()`
Cannot be intercepted from `TheView`, since parent inheritance is used in locating the method. An intercept could be installed in the base view. However this would cause the method to always be called with `self` pointing to the base view, even if the original message was directed at `TheView`. The method may or may not work right in this situation.

`TheView:Open()`
Cannot be intercepted. An intercept for the `Open` method could be installed in the root view, but it would catch every view opening in the Newton, and cause all of them to fail because the value of `self` would never be right. In other words, don't try this!

Options Available When Creating an Intercept

There are several options available when creating an intercept with the Install Intercept command:

Name for view

An arbitrary text string identifying the frame that the intercept is installed in. This is used in the debugging information shown when the intercept is triggered, to distinguish between multiple intercepts that you may have installed. Defaults to the `debug` slot in the frame, if any. Otherwise, tries to find a slot in a parent view that refers to the current view, which probably represents a declaration of the view. If no meaningful name for the view can be found, "unknown" is used.

Method to intercept

Allows selection of any method that can be intercepted from this frame. This includes all slots containing executable routines (objects of class `'CodeBlock`, `'CFunction`, and `'BinCFunction`) that exist in protos of the current frame, but that do not exist in the frame itself.

Type of intercept

Allows selection of various debugging actions that the intercept can take when triggered. These are discussed in the next section.

Conditional expression

An optional NewtonScript expression, or series of expressions separated by semicolons (in which case the value of the final expression is used), which controls the triggering of the intercept. If the expression evaluates to `TRUE` (or any other non-`NIL` value), the intercept performs its debugging action as specified. If the expression is `NIL`, the original method is called, just as if the intercept didn't exist.

The conditional expression can directly access (and even modify) the parameters of the method, by their actual names if they can be determined, or by "`arg1`", "`arg2`", and so on otherwise. If the expression includes a `return` statement, neither the specified debugging actions nor the original method are performed: the value given in the `return` statement (or `NIL`, if none is specified) is returned to the caller of the intercepted routine.

Keyboard button

Brings up or closes the Programmer's Keyboard, or hhe normal on-screen keyboard, if the programmer's keyboard is not installed.

Insert arg

Pops up a list of the parameters that the intercepted method takes, in the form which they can be used in the optional conditional expression. Tapping on one of them copies it into the expression at the current insertion point.

Install

Tapping this button creates an intercept routine according to the options specified elsewhere on this screen. If a syntax error is found in the conditional expression, an error notification is given, and the Interceptor screen stays open.

Close box

Tapping this returns you to ViewFrame without installing an intercept.

Intercept Types

All of the intercept types at least print to the Inspector some information about the method being called, in a form like this:

```
>>> view:method(arg1, arg2, ...)
<<< view:method returns value
```

The parameters and return value are displayed according to the current `printDepth` setting. You probably should have `printDepth` set to 0, or possibly even -1, to avoid being overwhelmed by the output.

WARNING: Some types of intercepts can put the Newton into a break loop when triggered. If this happens at a time when the Newton is not connected to the Inspector, there will be no way to give the `ExitBreakLoop()` command to continue normal operation, and the Newton will be completely locked up. You need to press the reset button to regain control of the Newton.

Here are descriptions of the actions performed by each intercept type. The symbol given for each type is used when installing an intercept under program control (this is discussed later).

Print (symbol: `'Print`)
Only the basic Inspector display of the method's parameters and return value is done. This is the default intercept type.

Trace functions (symbol: `'functions`)
Inspector tracing of function calls is enabled during the execution of the intercepted method, by setting the global `trace` variable to the value `'functions`. Two function calls appear in the trace output after the end of the method: `GetGlobals` and `Apply`. These are side-effects of turning trace mode off.

Trace all (symbol: `'trace`)
Inspector tracing of both function calls and slot accesses is enabled during the execution of the intercepted method, by setting the global `trace` variable to `TRUE`. This is normally an inadvisable setting, due to the enormous amount of output that is generated. However, you may find it more useful when the output can be limited to the execution of a single routine.

StackTrace (symbol: `'StackTrace`)
Just before calling the intercepted method, the debugging function `StackTrace()` is called. This prints to the Inspector a listing of the current hierarchy of active function calls. This intercept type doesn't work under newer Newton system versions, which apparently limit the `StackTrace` function to use within a break loop.

ViewFrame (symbol: `'VF`)
After the intercepted method is called, the intercept's `argFrame` is displayed in ViewFrame, replacing the current object path. (This does not currently work under Newton 2.0.) This frame includes the following slots:

- `_nextArgFrame`, `_Parent`, and `_implementor`, which are references to the method's context. `_Parent` holds the method's receiver (`self`).
- The parameters to the method, if any.
- `_VFresult`, a variable local to the intercept routine holding the method's return value. It does no good to change this or the parameter slots, since the method has already completed execution and returned to its caller before these slots appear in ViewFrame.
- Possibly some other slots, if the conditional expression for the intercept uses any local variables.

It is not a good idea to use this intercept type on a method that is called in a loop: execution

is tremendously slowed down as ViewFrame repeatedly rebuilds its display of the `argFrame`. Also, do not install an intercept of this type anywhere in ViewFrame itself.

BreakOnThrows (symbol: `'BreakOnThrows`)
The intercepted method is executed, with the global variable `BreakOnThrows` set to `TRUE` just before, and reset to `NIL` just afterwards. Any exception thrown inside the method puts the Newton into a break loop. Type `ExitBreakLoop()` in the Inspector to return to normal operation.

Break before (symbol: `'before`)
Just before executing the intercepted method, the debugging function `BreakLoop()` is executed. The context of the break loop is the intercept routine: all of the method's parameters are available for examination and modification, but not any of the local variables of the method. As a convenience, the expression "`ExitBreakLoop()`" is printed to the Inspector just before the break loop is entered. Placing the cursor on this line and hitting Enter leaves the break loop and continues with the execution of the method.

Break after (symbol: `'after`)
Just after executing the intercepted method, but before returning to its caller, `BreakLoop()` is called. The context is again that of the intercept routine: the method's parameters are available, although they do not reflect any changes to their values performed inside of the method. The local variable `_VFresult`, which holds the method's return value, is also available. If `_VFresult` is modified, the changed value is returned to the caller when `ExitBreakLoop()` is executed.

If you need to install an intercept that breaks both before and after calling the intercepted method, use the "Break after" Intercept, and enter the following as a conditional expression:

```
BreakLoop();
true;
```

Interceptor Internals

Intercepts can also be created under program control. For example, if you are having trouble with a particular method, you can install an intercept in the `viewSetupFormScript` of the view containing it, and remove the intercept in the `viewQuitScript`. This saves you the trouble of locating the view and installing the intercept manually every time you download a new version of your program.

The following two functions are available when both ViewFrame and VF+Intercept are installed (ViewFrame may need to be opened once to make them available, if it is installed on a different store than VF+Intercept), and are invoked by calling methods in ViewFrame's base view. A reference to the base view can be gotten with code like this:

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

`VF:?Intercept(`*`frame`*`, `*`name`*`, `*`method`*`, `*`type`*`, `*`cond`*`);`
Installs an intercept. frame is the location to place the intercept: this would be the current object when installing an intercept manually. name is an arbitrary string describing frame, as specified in the "Name for view:" field. method is the symbol of the method call to be intercepted (not a reference to the method itself), as specified in the "Method to intercept:" field. type is one of the symbols (`'Print`, `'trace`,and so on) as documented in the list of intercept types, above. cond is a string holding a conditional expression, or NIL if the intercept is to be unconditional.

Please note that no error checking is done by this call (see the beginning of this chapter for the conditions under which intercepts can be successfully installed). Call this method inside of a `try/onexception` block if there is any possibility of invalid parameters, or of syntax errors

in the conditional expression.

`VF:?Outercept(`*`frame, method`*`);`
Removes an intercept. frame and method are the same as passed to the original `VF:?Intercept` call. Currently, this call simply does a `RemoveSlot(`*`frame, method`*`)`. However, in future versions of the Interceptor it may also update internal information about the currently installed intercepts, so you should always use this call rather than doing a `RemoveSlot` yourself.

(NEW) VF+Dante
Version documented: 1.1

This package adds various features specific to Newton 2.0 systems ("Dante" was the code name for Newton 2.0 during its development). DO NOT INSTALL THIS PACKAGE ON NEWTON 1.x SYSTEMS!

Annotation: Virtual Binary Objects
Used: When the current object is a VBO (virtual binary object), as detected by the `IsVBO()` function.

Adds the note "• VBO info" to the top of the object display. Tapping on this line adds a new frame to the object path, containing the results of applying several informational functions to the VBO. The names of the functions, which are also the names of the slots containing their results, are `GetVBOStore`, `GetVBOStoredSize`, and `GetVBOCompander`.

Annotation: Entry Aliases
Used: When the current object is a soup entry alias, as detected by the `IsEntryAlias()` function.

Adds the note "• Alias to soup entry" to the top of the object display. Tapping on this line adds a new current object to the object path: the result of applying `ResolveEntryAlias()` to the object. This is the original soup entry that it references, or `NIL` if the soup entry no longer exists.

Annotation: Mock Entries
Used: When the current object is a mock soup entry, as detected by the `IsMockEntry()` function.

Adds the note "• Mock entry (tap for handler)" to the top of the object display. Tapping on this line adds a new current object to the object path: the result of applying `EntryHandler()` to the object.

Info Popup Item: Image of View
Available: When the current object is an open view.
Shown: A bitmap image of the view, to make it easier to identify an unknown view object. If the view is larger than 160 pixels in either dimension, the middle part of the image is omitted and replaced by wavy lines.
Tap action: None.

Known problems:

- If the view has the `vClipping view` flag set, the image has the same clipping applied as the view itself. In the worst case, the image may be completely blank: this

happens if you are looking at the base view of an application which is obscured by ViewFrame or other views. Views without `vClipping` set appear completely, even if they are a child of a clipped view.
- Tapping on the image may actually select another item in the info popup. This is a bug that has been present since the earliest Newton systems, but wasn't discovered until too late for it to be fixed in Newton 2.0. Apparently I'm the first person to try putting a tall, non-tappable bitmap into a popup.

Drop Tester

Available: Always.
Object path: No immediate change, but the object path will be replaced by a new object when a suitable object is dropped into the tester slip.

Opens a slip with features to aid for testing the new Drag & Drop feature of Newton 2.0. The entry line at the top specifies the type of dropped data that is accepted. You can enter any symbol, or tap on the label to choose from the common drop types of text, polygon, ink, picture, icon, or meeting. The box in the middle is the area that can accept the dragged item. Its border thickens when an item convertible to the specified drop type is dragged over it.

The radio buttons specify exactly how the drop is to be handled. The first two cause the drop area to indicate that it doesn't want the data, even though it is the right type. The source of the data should avoid deleting the original data in this case, but not all drag sources do this. In particular, the current Newton 2.0 clipboard always deletes the original data, unless you double-tapp on it to copy it as you started the drag.

The first radio button option causes ViewFrame to display the drag info array that is passed to various Drag & Drop routines – it is primarily meaningful to the drag source. The other options display the actual dragged item. If a single item is dropped it is displayed in ViewFrame as is, otherwise an array of the items is displayed.

If you don't need this much control over the drop process, note that ViewFrame's tiny palette form is also a drop target, accepting only the common data types.

Get Base View

Available: Always.
Object path: Replaced with new object.

Pops up a list of all frames in the root view (typically, application base views and built-in slips) which have a name associated with them. The name may come from an `appName`, `title`, or `debug` slot. Tapping on one makes the frame the new current object. This feature is intended to replace the Get App command in VF+General, which no longer works under Newton 2.0.

Get Part

Available: Always.
Object path: Replaced with new object.

Allows selection of any installed package part. First, you are given a pop-up list of all installed packages. Selecting one then gives you a pop-up list of all of the parts in that package. Most packages contain only a single form or auto part, but any number may be present (such as custom fonts used only by that package). Note that the resulting object is always a read-only reference into a package. This command cannot give you a reference to any RAM-based objects derived from a package, such as the base view of an application form part.

Soup Browser

Available: Always.
Object path: No effect (if no selections made in the browser), or replaced with new object.

Pops up a list of all soups present on the internal store (soups existing only on other stores are ignored), and builds a browser for entries in the chosen soup. The browser is based on the ListPicker prototype, used by the system for such purposes as choosing a person from the Names soup to send a message to. Note that the browser operates on a union soup, and therefore shows entries on all stores, but this command currently only allows selection of a soup if it exists on the internal store.

The action taken when the browser is closed depends on the number of entries which have been selected. If nothing is selected, the current object in ViewFrame is unchanged. If a single entry is selected, it becomes the new current object. If multiple entries are selected, an array of aliases to the entries becomes the new current object. The entry alias annotation feature in VF+Dante allows easy access to the aliased entries, without having to keep all of them in memory at all times.

The columns of the browser are derived from indexed slots of the soup, with preference being given to string-valued indexes. The index used in the left-most column is also used in the soup query. Any entries that lack this slot simply don't appear in the browser. Multi-slot indexes are supported. The values of all component slots of the multi-slot index are shown, separated by slashes. Tags indexes are currently ignored.

(NEW) BugDrop

Version documented: 1.0

BugDrop is a debugging accessory (not actually an Addition) for use with Newton 2.0 only. DO NOT INSTALL IT ON NEWTON 1.x SYSTEMS. It is usable only as the backdrop application: to set it as such, select its icon in the Extras Drawer then tap on Make Backdrop in the Action popup. If opened from the Extras Drawer, BugDrop simply closes itself.

As the backdrop, BugDrop's main purpose is to sit there and do nothing. There are two reasons why might want a backdrop application that does nothing:

- More free memory is available to debugging tools than if a "real" application is the backdrop.
- It gives you a clean background for doing screen shots. By default, BugDrop fills the screen with a light gray pattern: this can be changed to any of the five standard fill patterns by tapping the scroll arrows. All other applications that use the scroll arrows must be closed in order for BugDrop to receive the taps. You'll probably use a white background for screen shots, but this is not the default since it would make the Newton appear to be turned off if no other applications were open, causing great confusion.

Actually, BugDrop does do something other than just sit there: tapping anywhere in the screen background turns off trace mode, and pop up a list of debugging options. The popup only appears if ViewFrame is installed. The first option opens ViewFrame, or brings it to the front if it is already open. The second item opens the NTK Toolkit App if it is installed. The

`toolboxSym` slot in ViewFrame's base view can specify a different application to open, just as it does for the Toolbox button in ViewFrame itself.

The remaining options are a subset of the installed Addition commands, including only those commands which do not require a current object to operate. Commands with names starting with "About" are omitted from the list. Selecting an item works just the same as if the command is selected from the Additions popup in ViewFrame. ViewFrame automatically opens if the command requests the display of an object.

Since BugDrop has no buttons or other controls and does not display anything but a blank screen, there is no direct way to tell what version you have installed. This information is contained in BugDrop's base view (`GetRoot().|BugDrop:JRH|`), in a slot named `version`.

VF+Sample

Version documented: 1.0

This package includes a few mostly useless Additions. Complete NTK source code is provided on disk. It is intended as an example of how to use the Addition mechanism, and as a base for user-written Additions (see the next section for details). The supplied project was created by NTK 1.6, and should be compatible with NTK 1.5 as well. If you need a version that works with older versions of NTK, please contact the author, Jason Harper (contact addresses are in the ViewFrame manual).

Huge Bitmap/Picture

Available: If the current object is a bitmap or a picture (the name shown for the command reflects the type of object).
Object path: Unchanged.

Displays the object in a separate view, scaled to fill the entire screen.

Viewer: 42

Used: If the current object is the number 42.

Displays a special message related to this number. If you don't understand the message, you need to read more Douglas Adams.

Annotation: Integers (Even/Odd)

Used: If the current object is an integer.

Tells you whether the integer is even or odd.

Info Popup Item: Current Time

Available: Always.
Info shown: The current time.
Tap action: The current time, as returned by the `Time()` function, becomes the current object.

Writing Additions

The material in this section Supercedes everything in the "Writing Add-Ons for ViewFrame" section of the Advanced Uses chapter of the ViewFrame 1.0 manual. The add-on mechanism described there, based on appending items to ViewFrame's routing frame, turned out not to work very well in practice. The problem is that frames with more than 20 slots get sorted so that slots can be located faster. It doesn't take too many add-ons to put the routing frame over that limit, completely scrambling the contents of the Action menu.

Additions written to this specification work with ViewFrame version 1.2, and all future

versions. Version 1.0 ignores them. Version 1.1 supports most of the Additions interface given here: all specific cases where a feature wasn't present in 1.1 are indicated. A complete NTK project for a sample Addition called VF+Sample, which you can use as an example and as a base for writing your own Additions, is supplied on disk.

Much of the sample code here assumes the existence of a variable named `VF`, which holds a reference to ViewFrame's base view. Many of the routines related to Additions are passed `VF` as a parameter: if it is needed elsewhere, it must be explicitly initialized with code such as:

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

Starting with ViewFrame 1.2, ViewFrame's version is contained in the `version` slot in its base view, in the form of a real number. Your Additions may need to check this slot if they use any features that are version-specific. The `version` slot didn't exist in earlier versions of ViewFrame, so merely checking for its existence is the same as checking whether it is greater than or equal to 1.2.

ViewFrame Additions are contained in "Addition packages", which are typically autopart packages (note that the sample code I provide won't work right as a normal package). Each Addition package contains an `Addition` list, which is an array of one or more `Addition` frames, each of which defines one or more add-on features. A frame of references to the Addition lists from all installed Addition packages is contained in the global variable `GetGlobals().|ViewFrame:JRH|` (this is the same symbol as ViewFrame's app symbol). An Addition package's `InstallScript` should add its `Addition` list to this frame (initializing it to an empty frame if it doesn't already exist), using its app symbol as the slot name. Likewise, the package's `RemoveScript` should remove its list from the frame, and remove the frame itself if it becomes zero-length. ViewFrame itself only looks in the global variable to find its Additions: it never creates or modifies the frame.

Each type of Addition requires specific slots to be placed in an Addition frame. Different types of Additions use different slots, so you can place multiple Additions into one frame as long as they are all of different types. Multiple Additions of the same type require multiple frames in the `Addition` list. Do not place any other slots in an `Addition` frame than the ones documented here: they may conflict with new types of Additions defined in the future. If you need to make additional slots available to your Addition routines, place them in a frame referenced by a `_parent` (not `_proto`) slot in the `Addition` frame: the slots will be accessible by your routines, but ViewFrame won't see them.

All functions in `Addition` frames are called as methods in the context of the Addition frame containing them: therefore, they have direct access to other slots in the frame, and to any additional context that you give them via a `_parent` slot in the frame. Note that this context is not a view context: any method calls that require a view context (such as `:Notify()` or `:SysBeep()`) must be preceded by `GetRoot()` or `VF` in order to send them to a suitable context.

Commands

Commands are Additions that appear in the Additions popup in ViewFrame, and are invoked directly by the user. They may appear in the popup either unconditionally, or only when the current object is of a particular type. A dividing line is automatically added between commands belonging to different Addition packages. By convention, the first command in every package should be titled "About package name", and should bring up a box giving a copyright notice and a list of other Additions contained in the package. See the sample code for an example of how to do this.

Do not give any other commands a title starting with "About". This causes them not to appear in the normal list of commands. Command titles are typically strings, but they can be any valid data item for a picker list. Bitmaps are another possibility, and you may find use for the ability to specify a mark character for the item.

Required Slots

Addition frames for a command must have a `title` or `GetTitle` slot (but not both), and an `AdditionScript` slot. They are defined as follows:

`title`
For unconditionally appearing commands, this slot contains the title of the command (typically a string or bitmap). The command always appears in the Additions popup, regardless of the type of the current object, and even if there is no current object. The command also appears in the popup produced by the BugDrop accessory.

`GetTitle: func(obj)`
For conditionally appearing commands, this routine is called whenever the Additions popup is opened to see whether the command should appear. The routine should return either a value suitable for use as a title, or `NIL` if the command is currently inappropriate. The `obj` parameter contains ViewFrame's current object, or `NIL` if there is no current object. There is no way provided to distinguish between there being no current object, and the current object having a value of `NIL`.

`AdditionScript: func(obj, VF)`
When this command is selected by the user (which can only happen if a `title` slot is provided, or the `GetTitle` routine returns a non-`NIL` value for the same `obj`), this routine is called to perform the command. `obj` is again ViewFrame's current object or `NIL`, and `VF` is a reference to ViewFrame's base view.

Useful Methods

Some methods that may be of use in the `AdditionScript` routines of commands (some of which are documented in the manual, but are repeated here for completeness):

`VF:NewEval(`*`expr`*`);`
`expr`, which must be a string, is compiled and executed. The result becomes the new current object, with *`expr`* itself used as the description in the object path. The previous object path is discarded.

`VF:NewValue(`*`text, value`*`);`
The previous object path is discarded. *`value`* becomes the new current object, with *`text`* as its description in the object path.

`VF:AppendEval(`*`slot`*`);`
A new object is added to the path, derived from the current object (if there isn't one, this method does nothing). The exact behavior is based on the class of the *`slot`* parameter:

- Symbol. A frame slot in the current object (which had better be a frame) becomes the new current object. The text added to the object path description is "*`.slot`*".
- Integer. An array element from the current object (which had better be an array) becomes the new current object. The text added to the object path description is "*`[slot]`*".
- Array of class `'pathExpr`. An arbitrary path expression is applied to the current object to produce the new current object. The text added to the object path description is the concatenation of "*`.symbol`*" or "*`[index]`*" items for each element of the path expression.
- Frame, containing slots named `text` and `value`. `value` becomes the new current object, and `text` is added to the object path description.

VF:AppendValue(*text*, *value*);
value is added to the object path and becomes the new current object. *text* is added to the object path description. If no current object exists, this is the same as :NewValue().

VF:ShowResult();
Rebuilds the display of the current object. You would use this if your command modified the current object rather than adding a new object to the path, or if it enabled or disabled a viewer that applies to the current object.

Viewers

Viewers are Additions that automatically display particular types of objects in more detail than provided by ViewFrame's built-in object viewers. A viewer entirely replaces the normal object display with its own display. The results should go in ViewFrame's normal object display area. An object displayer that uses a separate view should be implemented as a command Addition rather than a viewer.

Tapping the Fmt button and selecting the Alternate format bypasses all viewer Additions, and always uses a built-in object viewer (and may have other effects on the display of the object, as described in the ViewFrame manual).

Addition frames for a viewer require only a DisplayObject slot. This can go in the same Addition frame as a command's slots. Neither interferes with the other, and you save a slight amount of memory over using separate frames.

Calling a Viewer

Whenever ViewFrame is about to display a new or changed object, and the Normal display format is selected, it calls any DisplayObject routines using

```
DisplayObject: func(obj, VF);
```

in all Addition frames in order, allowing each to override the object display. If a DisplayObject routine returns TRUE, no further DisplayObject routines are called, and the built-in object display is not called. If all routines return NIL, the built-in object display is called.

obj is ViewFrame's current object (which is guaranteed to exist: no DisplayObject routines are called when there is no current object to display). VF is ViewFrame's base view.

Utility Functions

Several routines are available to DisplayObject routines for adding items to the object display. Note that these routines are not valid for you to call at any other time. DisplayObject routines must not call any of the object path-changing calls documented in the commands section of this document.

VF:AddStatic(*text*);
Adds a one-line static text item to the display.

VF:AddPara(*text*);
Adds a multi-line, word-wrapped text item to the display.

VF:AddIcon(*icon*);
Adds a bitmap (frame with bounds, bits, and possibly a mask slot) to the display. Always includes a static text item giving the icon's size, followed by the icon's normal image. If a mask is included, the display also includes the mask, plus the icon in its hilited state.

VF:AddPict(*icon*);
Adds a picture (basically, anything that can go in the icon slot of a clPictureView,

although the VF:AddIcon call is better for bitmaps) to the display.

VF:AddSound(*snd*);
Adds a sound frame to the display, and plays the sound.

VF:AddInt(*num*, *sym*);
Adds the integer *num* to the display, in decimal and hexadecimal form. The *sym* parameter should be either NIL or a symbol. If it's a symbol, the integer is also shown in symbolic form using constants appropriate for a slot named by that symbol, and there is be a pop-up menu of other slot symbols for which constant values are known. Values of sym for which constants are known currently include 'viewJustify, 'viewClass, 'viewFlags, 'entryFlags, 'viewFormat, 'viewEffect, 'viewTransferMode, 'viewFont, 'textFlags, 'numArgs, and '|Key Descriptor| (key descriptor values are normally in an array rather than a frame, so there is no real slot symbol for them).

VF:AddSelect(*text*, *slot*);
Adds a tappable item to the display, as used in the normal frame and array viewer. *text* defines the appearance of the item. It should start with a space so that the item doesn't run into the left edge of the display area. *slot* is passed to the VF:AppendEval method when the item is tapped. See the documentation on that method (in the Commands section of this document) for details on what slot can be: basically, a symbol, integer, path expression, or frame containing text and value slots.

The above methods cover all of ViewFrame's current object display capabilities, with the exception of hex dumps. The internal representation of hex dump items has changed a few times over the life of ViewFrame, and is likely to change again. I'm not willing at this time to commit to a format for hex dump items that I will guarantee in future versions. If you have a need for displaying hex data in a viewer you are writing, contact me. I may have settled on a permanent format by then, or I may be willing to guarantee future support for hex dump items with particular characteristics (such as being below a specific size). Likewise, I am not currently prepared to support any way of adding entirely new item types to the object display. Contact me if you need this ability, and we'll see what we can work out.

(NEW) Annotations
Annotations are Additions that add to the top of the object display additional information, whether it was generated by a built-in viewer or an Addition, and whether or not the Alternate display format is selected. This is a new feature in ViewFrame 1.2. Previously, annotations were handled by viewer Additions that returned NIL from their DisplayObject script, allowing other viewers a chance to claim the object. The problem was that a real viewer that happened to load before the annotation Addition would prevent the annotation from working. Therefore, these two types of object displaying Additions have been split into separate scripts, so that they cannot interfere with each other.

Addition frames for annotations require only an AnnotateObject slot. All AnnotateObject scripts in all Addition frames are called whenever an object is displayed.

AnnotateObject: func(obj, VF);
The parameter values, and the base view methods available for use in the script, are exactly the same as for the DisplayObject script described earlier. However, the return value of the AnnotateObject script is ignored.

Info Pop-up Items
These Additions are called whenever the object description line is tapped. They may add any number of items to the resulting pop-up menu. Since they are called only on demand, they are suitable for information items that may take a long time to generate, and therefore aren't

appropriate as an annotation that is generated each time the object is displayed.

A single slot is required in an Addition frame:

```
AddInfo: func(obj, addtap, addnontap)
```

`obj` is ViewFrame's current object. It is guaranteed to be a non-immediate value, since the info popup is not available when the current object is an immediate value, or when there is no current object.

`addtap` is a closure which can be called to add a general item to the popup:

```
call addtap with (item, value, desc);
```

`item` is the item to be added to the popup. It can be anything that a standard picker can display. *`value`* and *`desc`* are passed to the `VF:AppendValue` method (in the opposite order) when the item is tapped. They should be set to `NIL` if item is non-tappable.

`addnontap` is a closure for adding an item that isn't tappable:

```
call addnontap with (item);
```

It is simply a shortcut for doing this:

```
call addtap with ({item: item, pickable: nil}, nil, nil);
```

Note that some picker item types cannot put entries into an item frame like this. Such items, including separator lines, should be added with `addtap`.

Other Addition Routines

There are t other slots currently defined for use in Addition frames:

`VFSetupDone: func(VF)`
Called whenever ViewFrame is opened.

`VFQuit: func(VF)`
Called whenever ViewFrame is about to be closed. One possible use for these slots is to install some patch to ViewFrame while it is running, and to remove it afterwards.

If your Addition needs to maintain some persistent data, you may store it as a single slot (which can, of course, be a frame containing as many slots as you need) in ViewFrame's base view. Use your Addition's app symbol (the same symbol you use to store your Addition list in the Additions global variable) as the slot name, to avoid conflicts.

Other ViewFrame Internals

Finally, here are a few other ViewFrame methods that may be of use in Additions or other code that interacts directly with ViewFrame (also see the Advanced Uses chapter of the manual).

All Versions of ViewFrame

`VF:oneliner(`*`parent, slot, value`*`)`
Returns a brief textual description of the object *`value`*, as used in ViewFrame's normal frame and array listings. If *`parent`* is not `NIL`, and `value` is equal to `parent`, its description is given as "SELF". *`slot`* is only meaningful if `value` is an integer. If `slot` is a symbol for which ViewFrame knows constant values (see the `VF:AddInt` method above), and `value` matches one of those values, its description also includes the name of that constant. Set

parent or slot to NIL if you don't need their respective features.

VF:Print(*value*);
Writes *value* to the Inspector followed by a carriage return. Not terribly useful at the moment, but future versions of ViewFrame may implement alternatives to Inspector output (such as sending the data out the serial port, or displaying it onscreen), and by using this method rather than the Print() global function, your app or Addition will automatically take advantage of any such new features.

VF.exprLine:FirstExpr(*text*);
VF.exprLine:NextExpr(*text*);
FirstExpr adds a dividing line and the expression specified by *text* to the expression pop-up list. NextExpr adds additional expressions to the list. It can be called as many times as you want, but FirstExpr has to be called once before NextExpr works. Added expressions are lost whenever the current object changes. An appropriate place to make these calls would be in a AnnotateObject script (or a DisplayObject script that returns NIL, in ViewFrame 1.1). These routines might have no actual effect on object display other than adding expressions relevant to some particular kinds of objects.

If an expression you add has a place where a string value needs to be specified by the user (like the standard GetUnionSoup(" ") and debug(" ") expressions), you should include a pair of double quotes separated by a space. When such an expression is selected from the pop-up, the space between the quote marks is deleted, and the insertion point is placed at that location for easy entry of the string.

ViewFrame 1.2 or Higher
The following methods are only available in ViewFrame 1.2 or higher:

VF:StdArgFrame(*fn*)
Creates a standardized form of the argFrame of function fn, taking into account the differences in format between Newton 1.x and 2.0 functions. The function's DebuggerInfo slot, if present, is used to extract actual parameter and variable names. The result is an array of symbols: it is constructed each time, so you can safely modify the array.

The first three elements are always '_nextArgFrame, '_parent, and '_implementor. The following elements are the function's parameters, if any, with names of the form Arg*n* assigned if the actual names cannot be determined. The remaining elements are the function's local variables, with names of the form Local*n* assigned if actual names are not available.

To see an example of this method at work, look at the Insert Arg button in VF+Intercept's Add Intercept slip. The generation of items in the pop-up begins by applying VF:StdArgFrame() to the selected function. The first three elements (internal system variables) are removed from the array, then the array's length is set to the number of parameters (the low byte of the function's numArgs slot) to get rid of the local variables at the end.

VF:StdObjDesc(*frame*)
Returns a brief textual description of a frame, based on slots such as debug, text, name, and so on, or NIL if no meaningful slots are found. If a string is returned, it always starts with a space. This might be useful in cases where VF:oneliner() returns too much information.