-=[VIEWFRAME 1.3ß2]=-

• ViewFrame is now based on a resizable floater proto provided by Rob Bruce and Michael Krzyzek of Tactile Systems. The view can be resized by dragging any corner or side, except for the top center which is now the only place where you can drag the view itself.

• The Eval button is now declared as the default button, so you can evaluate an expression just by hitting Return (hard or soft keyboard). However, this only works if the expression entry area is 1 line high: if it's expanded, Return simply inserts a carriage return. To execute a multi-line expression from the keyboard, see the VF+Keys add-on.

• There is now a quick alternative to the ViewFinder for selecting a currently open view: just drag from VF's minimize button (the one to the left of the close box) to any view. To cancel a selection in progress, drag back to near the button (so that the selection pointer disappears) and lift the pen there. The selected view becomes the new current object, replacing the existing object path.

If the selected view contains a method named `|GetValue:ViewFrame:JRH|`, it is called (with no parameters), and the result is used instead of the view itself. The idea is that you can define this method in some view that you would otherwise never need to select (your close box, perhaps), and have it return some significant debugging value: your routing target, an endpoint, a state frame, etc. Examining this value is now a matter of a single drag, rather than digging through the view hierarchy to find it.

• The space formerly taken by the resize buttons (now obsolete) is now used for two Saved Object buttons. These buttons can be used to save arbitrary objects, and redisplay or perform other actions on them later. They can also be used for Additions that work on multiple objects (imagine a displayer that shows two objects side-by-side, with differences hilited), but nothing like that is currently implemented.

The options in the popup you get when tapping on a saved object are:
- "No saved object", if there isn't one.
- A brief description of the object, if there is. Tapping on this item does the same thing as the next...
- "View (append)" displays the object, appending to the current object path.
- "View (new object)" displays the object, replacing the object path.
- An option to call the object, if it is a function taking 0-2 parameters. This option allows easy definition of temporary commands that can be used repetitively. The behavior depends on the number of parameters:
-- "Call with ()" - result replaces object path.
-- "Call with (obj)" - called with current object (or NIL if there isn't one), result appended to path.
-- "Call with (obj, VF)" - the 2nd parameter is a reference to ViewFrame's base view. In this case, the result is not displayed: the function can call any of VF's object display methods it wants, or none at all if that is appropriate to its action.
- "Paste reference" - a NewtonScript reference to the saved object is inserted into the current key-receiving view as if typed on a keyboard. If this is VF's expression entry area, the reference is of the form "VF.Saved[*n*]": this works because all evaluated expressions now have a variable VF available to them referencing VF's base view. If pasted into any other view, the reference is of the form "GetRoot().|ViewFrame:JRH|.Saved[*n*]", which will work regardless of context. The reference can be used in an expression, or assigned to.
- "Exchange A & B" swaps the two saved objects (even if one doesn't currently have anything saved in it). This is intended for use with Addtions that use the saved objects as parameters, and might assign different meanings to A and B: it's not very useful at the moment.
- "Clear" removes VF's reference to the saved object, so that it can perhaps be garbage collected.

- "Save current object" appears if VF has a current object (and it's not NIL).
- "Save as [A, B]" also appears if the current object is a 2-element array: this is a shortcut for defining two saved objects at once (perhaps two command functions).
- "No current object" appears if there isn't currently an object that can be saved.
- Additional options may be added to this popup via Additions (details below).

Whenever there is a saved object, the following actions occur:
- The button's name appears as a capital letter, to remind you that an object has been saved.
- Whenever VF's currently displayed object is one of the saved objects, an annotation of the form "This is saved object A" will appear. This check uses exact identity, not the normal "=" test which can be true for distinct objects of certain classes (symbols and reals).
- TODO: If the current object isn't identical to any saved object, but shares its frame map with one, this will eventually be indicated as well.
- If any slots in the current object are equal to a saved object (normal "=" testing in this case), a note of the form "= A" will be appended to their descriptions. This feature is provided by the `VF:oneliner()` method, and works for Additions as well. To disable it in places where it is inappropriate (perhaps it's obvious from the context that the object being described is one of the saved objects), pass the symbol `'Saved` as the *parent* parameter to `VF:oneliner()`.
- You can drag from the saved object button to select a view, just as with the minimize button. This currently only does anything if the saved object is a function taking 1-3 parameters, and again the behavior depends on the number of parameters:
-- 1 parameter - function called with the selected view (or result from *`view`*`:|GetValue:ViewFrame:JRH|()`) as the parameter, result replaces the current object path.
-- 2 parameters - called with (view, VF), result not displayed (unless the function uses one of VF's object display methods).
-- 3 parameters - called with (view, VF, obj); 3rd parameter is VF's current object, or NIL if none.
Here's something to try with this feature. Evaluate this function:
```
      func(v,vf) v:hilite(band(getviewflags(v), 33554432)=0)
```
and store it in one of the saved object buttons. You can then drag from that button to any view to toggle its hilite state. Apart from being fun (ever tried hiliting the root view?), this might be useful for taking screen shots with a button hilited as if though it was being tapped.

• Saved object internals: the saved objects are elements of an array called `Saved` in VF's base view. Currently the array contains 2 elements, but this may be expanded in the future (perhaps more saved object buttons would be visible depending on VF's current width). There will never be more than 26 elements, so they can always be described by single letters. Element `0` is saved object A, element `1` is B, etc: you can get the letter with the expression `chr(ord($A) +` *`index`*`)`. Additions can lengthen the array even now: the annotations added to objects and slots that are equal to a saved object will work with the added elements. However, if you do this make sure you also provide a way to clear the extra elements, as there is no user interface in VF for doing so.

If you modify the `Saved` array, you can call `VF:UpdateSaved()` to update the saved object buttons (in particular, it changes the capitalization of the button names to indicate the presence of saved objects). This is not necessary for saved object popup Additions, or any Additions which also display a new object, as the buttons are automatically updated at those times.

It is possible that an Addition command could be called at a time before VF has created the `Saved` array (such as by tapping on BugDrop's background if VF hasn't been opened since the last reset). In this case the Addtion will need to create the array itself: here's suitable code for doing so.
```
      if not IsArray(VF.Saved) then VF.Saved := array(2, nil);
```

To add items to the saved object popup, put the following slot in any Addition frame:

```
      AddSaved: func(index, obj, cur, VF, add1)
```

- *index* - the index of the tapped-on object in the `Saved` array.
- *obj* - the saved object itself.
- *cur* - VF's current object, or NIL if none.
- *VF* - VF's base view
- *add1* - a closure which can be called to add an item to the popup, used as follows:
-- `call` *`add1`* `with (`*`item`*`,` *`actionFunc`*`)`
-- *item* - a string or other item suitable for use in a standard picker.
-- *actionFunc* - a function that will be called if the item is selected, or NIL if the item is unselectable.
The *actionFunc* is called with no parameters: it should be a closure defined inside your `AddSaved` function, which will let it inherit any needed values from that function's parameter list.

If you add any items to the popup, the first one should be a `'pickSeparator` item to divide it from the built-in items.

• There are two new kinds of filter expressions which can be used when the current object is a frame, to go along with the existing `=`*`text`* expression that limits the display to only slot names containing the specified *text*. They are:

```
      &symbol
```

Limits the display to only the slots whose class exactly matches the *symbol* (no substring or subclass matches). Example: use `&CodeBlock` (Newton 1.x) or `&_function` (Newton 2.x) to show only the bytecode functions in the current frame.

```
      #number
      #<number
      #>number
```

Limits the display to frames or arrays containing either exactly, less than, or more than the specified number of elements. Note that this conflicts with the `#`*`hexnumber`* form of reference specification, however that form only works on Newton 1.x devices. If you're still using it on an early Newton, you'll need to enclose it in parentheses.

• Display of frames or arrays with more than 150 slots is suppressed, since they are likely to crash VF. The exact threshhold can be set by changing the `MaxObjectSize` slot in VF's base view. You can select Normal (or other format) from the Fmt popup if you wish to attempt displaying the object anyway. Expressions and Additions are still fully usable on the object: a couple of particularly useful things to do with huge objects are filter expressions (see above), and the new Split Into Chunks command in VF+General.

• Addition commands with names ending in "Prefs" are now collected into a single "Prefs..." item at the end of the popup, similar to how commands starting with "About" are now handled.

• The sorting options in the Fmt menu are now much better behaved. They now only sort within groups of tappable items: static text and other special display items are left where they are. Also, leading dashes are ignored when sorting by slot: when using Alternate format to view a frame deeply, this means that the slots sort alphabetically regardless of the depth at which they were found.

• VF now knows how to display 2.1-style icons that include color/grayscale data.

• VF now handles 2.1 sound frames that use a <fixed:#4> object instead of a number for the sampling

rate. A "Play" button is added so that you can easily replay the sound.
TODO: Use the new protoSoundFrame methods to display more accurate details that include the effects of compression and sample size.
TODO: Recognize sounds that consist of a binary object only (class 'TDTFMCodec for the new sound synthesizer, for example).

• VF no longer thinks that the protoSoundFrame object itself is a playable sound.

• VF no longer throws when viewing a VBO of class `'text`, such as found in NewtWorks word processor soup entries. The problem is that `IsPrimShape()` was returning TRUE for such objects (apparently it simply checks the object's class), but of course trying to display one as a shape doesn't work.

• New versions of various base view object display methods have been added, taking a new options parameter:

```
	VF:AppendEval2(slot, options);
	VF:AppendValue2(text, value, options);
	VF:NewEval2(expr, options);
	VF:NewValue2(text, value, options);
```

The original versions simply call the new versions, passing NIL for the options. The old versions are NOT going away, and are appropriate for most uses: the new versions just give some extra flexibility.
The options parameter should be one of the following:
-	NIL - Use the default format for object display. This acts just as ViewFrame previously did.
- *symbol* - Use the specified display format for the new current object, just as if a format other than Normal had been selected from the Fmt popup. See the description of the `format` slot, below, for allowable values.
- *frame* - Allows arbitary display options to be specified. Currently defined slots are:
-- `format -` Specifies the display format to use. Possible values:
--- `'normal`, `'alternate`, `'location` - specify the existing built-in formats.
--- `'none` - No attempt is made to actually display the object, but you can still use expressions and Additions on it.
--- NIL or missing - uses `'normal` if the object is of a reasonable size, `'none` otherwise. You should use this instead of explicitly specifying `'normal`, unless you know the object is reasonably small.
-- `sort`: Specifies a sorting option to be applied to the object display.
---	NIL or missing - No sorting done.
--- `'bySlot`, `'byValue` - The built-in sort options, available in the Fmt popup.
-- `IntSymbol`: If the object being displayed is an integer (or otherwise uses `VF:AddInt()` as part of its display), this symbol overrides the normal mechanism for determing which set of integer constants to use in interpreting the value. View any integer and look at its "View as" popup to see the meaningful values for this slot.

• The second parameter to `VF:AppendEval()` or `VF:AppendEval2()` can now be a string, giving an expression to be evaluated to produce the new object. All expression forms that work in VF's expression entry area are supported here: slot/element access, filter expressions, expressions starting with a slash and containing percent signs, etc.

• The frame format for the *slot* parameter to `VF:AppendEval()` or `VF:AppendEval2()`, or *value* parameter to `VF:AddSelect()`, previously only allowed `text` and `value` slots. Two more slots are now supported:
- `prefix` - if present, this is concatenated with the `text` slot to produce the description of the new

object in the path. The idea is that if you are adding many similar items with `VF:AddSelect()`, you may be able to save heap space by using a single `prefix` string for all items (perhaps in a _proto shared by all of the item frames).
- `options` - if present, specifies format options for displaying the object. However, a non-NIL *options* parameter passed to `VF:AppendEval2()` will override this slot: it's mainly of use with `VF:AddSelect()`.

• New type of display object for use by viewers and annotations: a button list. View a sound, or a cursor (with the new VF+General loaded) for examples. To add your own buttons:

```
	VF:AddButtons(buttonList);
```

buttonList is an array of frames containing two slots:
- `text` - name of button (which will be auto-sized to fit this text)
- `value` - arbitrary object that identifies which button was pressed. Generally, this should be a symbol that includes your registered developer signature, or a frame or array with such a symbol as its class.

Any number of buttons may be specified, but there is currently no auto-wrap if there are more than will fit within the width of VF's display.

When a button is pressed, the following function is called in all Addition frames in order to handle it:

```
	ButtonHit: func(value, obj, VF)
```

- *value* - the value slot from the pressed button.
- *obj* - VF's current object (guaranteed to exist, since otherwise there wouldn't be any buttons to press).
- *VF* - VF's base view

The `ButtonHit` function should return non-NIL if it recognizes the value, so that further `ButtonHit` functions do not need to be called. Note that there is no requirement that the `ButtonHit` function be in the same Addition frame as the viewer or annotater that originally added the button: they could even be in different Addition packages. This mechanism was used, rather than directly referencing a function to handle the button press, to avoid sucking your Addition package into the Out Box when VF's display is being printed.

If no `ButtonHit` function handles the button, there is currently one built-in handler. If the button's `value` is an array, the global function specified by the array's class is called, with the elements of the array as its parameters. Any instances of the character constant `$%` in the array are replaced with references to VF's current object. For example, the value specified for the "Play" button added for sound objects is:

```
	[PlaySoundSync: $%]
```

which effectively does a `PlaySoundSync(object)` when the button is tapped.

• New base view method:

```
	VF:OpenViewFinder(size)
```

Opens the ViewFinder window, closing the main VF window if it is open. If the *size* parameter is non-NIL, you get the expanded ViewFinder window, otherwise the minimized (3 buttons) window.

• New recognized integer constants:
	Endpoint opCodes (opSetNegotiate, etc.).
	Endpoint state (kUninit thru kOutLstn).
	modifiers, as found in Newton 2.1 key command frames (also works with the key parameter to methods such as `viewKeyDownScript`).
	New 2.1-specific values for textFlags, primarily related to key handling.

• Selecting from the "View As" popup of a displayed integer no longer loses annotations added by an Addition (important due to the new cursor viewer and Make Frame/Array Cursor command in VF+General, which can result in an annotation being added to ANY object).

• Integers in the range produced by PackRGB (0x10RRGGBB) are displayed as percentages of red, green, and blue (or of gray, if all three components are equal).

• The view formats in the top half of the Fmt popup will be extensible via Additions. For many object viewers, of a specialized nature rather than being the natural way of viewing the object, this would make more sense than the existing DisplayObject mechanism. DisplayObject automatically replaces the default display of suitable objects, unless you implement a mechanism of your own for turning it on and off: the new view format mechanism is entirely under user control (or program control, by using the options parameter on the new versions of VF's object display methods). The new mechanism is not available yet (it's closely tied to some other changes I'm making to the Additions mechanism), but here are the basic details:

View format Additions will use a frame with the following slots:
- `title` - a string or other picker item describing the format.
- `symbol` - a symbol (based on your developer signature) uniquely identifying the format. This symbol can be used as the *options* parameter (or `format` slot in an *options* frame) passed to any of the new object display methods, to specifically request this format.
- `CanDisplay: func(obj)` - must return non-NIL if the object can be displayed in this format. This function must not have any side-effects: it will be called when building the Fmt menu, and each time the object is displayed (if the object becomes unsuitable for the view format, the `'normal` format will be automatically used instead).
- `AnnotateObject` - if non-NIL, all AnnotateObject scripts will be given a chance to add an annotation prior to the main object display. This should be TRUE for most formats: the only reasonable exception I can think of is a format for viewing damaged objects, which most AnnotateObject scripts would choke on.
- `DisplayObject` - if non-NIL, DisplayObject scripts are given a chance to display the object: if any return a non-NIL value, your FormatFunc won't be called. I imagine that most add-on view formats would set this to NIL, but a very generic viewer might set it to TRUE to defer to more specific viewers.
- `FormatFunc: func(obj)` - your actual display routine. It works exactly as existing DisplayObject scripts do, except that the return value is ignored.

-=[PROGRAMMER'S KEYBOARD 1.2ß1]=-

• Keyboard is wider, and much easier to use, if the screen width is at least 320 (2.0 device in landscape, 2.1 device in any orientation).

• Keyboard now includes Control, Command, and up/down arrow keys. None of these do anything under Newton 1.x. The arrow keys work under 2.0, but the new modifier keys are unusable: they don't stick down, so it's impossible to type another key with the modifier in effect. All of the new keys work under 2.1: you can actually use command-key equivalents from this on-screen keyboard! This allows you to test command keys on a MP2000 while connected to the Inspector, but it can also be quite handy: in particular, the ability to hit Cmd-A to select the entire expression entry area then copy or delete it. There are a few limitations to this ability:
- The thicker border on the view currently accepting command keys, and the lines above and below the default button, only appear if a hardware keyboard is connected. (TODO: See if overriding CommandKeyboardConnected() will fake the system into doing these without needing a hardware keyboard.)

- There is no way to use keyboard navigation in popup menus, since any tap dismisses the menu.
- Holding down the Command key doesn't bring up the keyboard help slip. TODO: provide some alternate way of doing this (perhaps via the keyboard's "?" menu, or a global key command in VF+Keys), so that the appearance of the help slip can be tested on a MP2000 with the Inspector connected. There may still be a problem if the slip contains enough items to be scrollable: I don't think it can be scrolled except via a hardware keyboard.

-=[VF+DANTE 1.2ß1]=-

• Fixed bugs in Soup Browser involving multi-slot indexes (this showed up on a MP2000 with one of the preinstalled NetHopper soups), and returning of entries which had been selected but then deselected.

• The Soup Browser now lists soups in its initial popup if they exist on any store, not just the soups on the internal store.

• If the new version of VF+General is installed, multiple-item selections in the Soup Browser are returned as a target cursor object, rather than an array of entry aliases. Combined with VF+General's cursor browser, this makes examining the results much easier: moving between selected entries now takes a single tap rather than 4.

• The Get Part command no longer throws an exception if you select a package containing any non-frame parts, such as the protocol parts in Newton Internet Enabler. This doesn't actually do any good as far as examining such parts: the only aspect of them accessible from NewtonScript is their name. However, it does allow you to examine any ordinary frame parts in the same package.

-=[VF+KEYS (no visible version number)]=-

This add-on (not actually an Addition) provides keyboard equivalents for many ViewFrame features. It only works on Newton 2.1 devices. Also, it must be installed in internal memory: allowing a card containing it to be ejected without triggering a Grip O' Death would require an enormous amount of EnsureInternal()ing. TODO: Add a DoNotInstall script to enforce these restrictions.

All of VF+Keys' equivalents require both the Command and Control keys to be pressed. This combination was chosen as a compromise between distinctness from existing key equivalents, and easiness to type. They are all global commands, available even in modal dialogs, although it may not be a good idea to open ViewFrame in front of an active modal dialog (you can always close ViewFrame via the keyboard if you hang the Newton this way). Here are the current key equivalents:

• V - Opens ViewFrame, if it isn't open. If it was open, brings it to the front. If it was already in front, places the insertion caret at the end of VF's expression entry area.

• G - Displays the frontmost application view (as returned by `GetView('viewFrontMost)`), opening VF if needed. Think 'G' for 'GetView'.

• K - Displays the current key-receiving view, as returned by `GetView('viewFrontKey)`.

• F - Opens ViewFinder directly, closing VF itself if it is open.

• Up and down arrows - Scrolls the VF display.

• Return - Equivalent to VF's Eval button.

• Left arrow - Equivalent to VF's Back button.

• + - Equivalent to VF's "+" (Additions) button. TODO: allow this command to work even if VF is not open, bringing up the same popup as tapping in BugDrop does.

• R - Equivalent to VF's Routing (Action) button.

• [and] - Shrink and grow VF's entry area.

• S - Equivalent to VF's Fmt button. Think 'S' for sort, since that's probably the most common use for this command.

• Z - Equivalent to VF's zoom button.

• A and B - Equivalent to VF's saved object buttons. TEMPORARILY DISABLED: these equivalents tend to hang the Newton, there seems to be a bug with keyboard-opened popups that start with an unpickable item.

• E - Opens the ViewFrame Editor, or other app as specified by the `editorSym` slot in VF's base view.

• T - Opens the Toolkit App, or other app as specified by the `toolboxSym` slot in VF's base view.

• N - Invokes the Step Into Next Button command from VF+Intercept, if installed (otherwise, it beeps). This allows easy use of the single-stepper to examine execution of any script triggered by a button press. The button press itself must currently be done by pen. It is intended that this feature also work with buttons pressed via a key equivalent, but that currently results in a hang.

• L - Toggles the backlight. Not really a debugging function, but I find it handy to be able to do this from the keyboard on my MP2000.

-=[VF+FUNCTION 1.2ß1]=-

• The presence of a null character constant ($\00) in a function no longer truncates the listing at that point. All characters under 0x20 now simply display a hex value, no attempt is made to insert the character itself as a comment.

• The function viewer now automatically handles native functions that include a bytecode version in their `bcFunc` slot: such functions are displayed starting with "func NATIVE(" rather than just "func(". It is assumed that the function in the `bcFunc` slot actually does the same thing as the native function, but of course there's no way to verify that. This change also fixes a bug with the display of bytecode functions that contain a nested native function, such as the ProtoClone method in the protoFSM sample code.

-=[VF+GENERAL 1.3ß1]=-

• New command: Split Into Chunks
Available: if the current object is a frame or array with at least 20 elements.
Object path: a new frame, consisting of smaller chunks taken from the original object, is added to the path. The size of each chunk is roughly equal to the square root of the original object's length, but never more than half of VF's object display size limit (default 150; specified by the `MaxObjectSize` slot in VF's base view). This command is primarily intended for use with objects larger than VF's display size limit, but can be used on smaller objects if it makes the desired elements easier to find.

If the original object was a frame, the chunks are split based on the slot names in alphabetical order. For example, the first chunk might be `|a-f|`, the next `|g-n|`, etc.

If the original object was an array, the size of the chunks is rounded to a multiple of 10 so that it's easier to tell what an item's index was in the original array. The frame containing the chunks will have numeric slot names, indicating the original starting index of each chunk. The names will be padded with zeros on the left so that all are of the same length: this ensures that they sort into their proper numeric order.

• New annotation: Cursor
Used: if the current object is a cursor or cursor-like object (defined as a writeable frame that implements the Entry, Next, and Prev methods).
Action: adds ":Entry()" and ":Clone()" buttons to the object display, which will perform the indicated action on the cursor and add the resulting object to the path. The Clone button is omitted if the object doesn't implement that method, or if it has already been cloned (determined by looking at the last expression in the object path). The Clone button is intended for use when examining a cursor that was retrieved from a slot or variable in an application, so that moving the cursor doesn't affect the application. There is no point in using the Clone button on a cursor generated in ViewFrame, such as by the Soup Browser in VF+Dante.

• New annotation: Cursor Entry
Used: if the current object was derived from the previous object by applying the expression ":Entry()" to it. Typically it would be done by tapping the Entry button added by the Cursor annotation described above, but it could be done by actually evaluating that expression in the entry area, or by an Addition that used ":Entry()" to describe the action it took.
Action: adds "First", "Prev", "Next", and "Last" buttons to the display, which will apply the indicated movement to the cursor object (the next-to-last object), and update the current object with the result of calling the `:Entry()` method of the cursor. The length of the object path doesn't change, so you can use these buttons as many times as you want without building up any unnecessarily referenced objects in the frames heap. Also, if the cursor object was generated by the Make Frame Cursor or Make Array Cursor commands (see next command description for details), the display will include the cursor's `curSlot` or `curIndex` slot, respectively, so you can tell where you are within the object being browsed.

The exact actions of the four added buttons are:
- First - Does a cursor:Reset() if defined, otherwise cursor:Move(-99999) if defined, otherwise repeats cursor:Prev() until the result is NIL.
- Prev - Does a cursor:Prev().
- Next - Does a cursor:Next().
- Last - Does a cursor:ResetToEnd() if defined (only implemented in Newton 2.0 for actual soup cursors), otherwise cursor:Move(99999), otherwise repeats cursor:Next() until it returns NIL.

• New command: Make Frame/Array Cursor (title depends on class of current object)
Available: if the current object is a frame or array.
Action: adds a cursor-like object to the object path which will iterate over the elements of the current object (in numeric order for arrays, in alphabetic order by slot name for frames). In conjunction with the Cursor and Cursor Entry annotations described above, this command will let you easily browse the elements of an object. The returned object implements only a small subset of a real cursor's methods, and isn't usable with most cursor-related system functions (like `MapCursor()`).

The returned cursor object will contain a `curSlot` (for frames) or `curIndex` (for arrays) slot that indicates the current position in the object being browsed. Unlike the slots in a real cursor, you are allowed to change this slot, which will take effect as soon as you tap the Entry button.

Frame cursors returned by this command behave differently than normal cursors in one respect: it is impossible to move past the first or last slot in the frame. Trying to do so will return the same slot value, rather than returning a NIL entry.

• New objects: `'kFrameCursor` and `'kArrayCursor`
The cursor prototypes used by the Make Frame/Array Cursor command are available for use by other Additions. To allow you to get references to such objects, there is a newly defined mechanism based on the `VF:MapAdditions()` method. Call this method with `'GetObject` as the first parameter, an array containing the symbol indicating the desired object as the second parameter, and TRUE as the final parameter: if an Addition implementing the desired object is present, the object will be returned, otherwise NIL. For these cursor prototypes, call their `:New()` method, passing the object to be iterated over as the only parameter, to get the actual cursor. Here is some sample code showing how to make a cursor out of a frame called `obj`:

```
    local protoCursor := VF:MapAdditions('GetObject, '[kFrameCursor], true);
    if protoCursor then local cursor := protoCursor:New(obj);
```

-=[VIEWFRAME EDITOR 1.2ß1]=-
• If both NS Debug Tools and the single-step capable version of VF+Intercept are installed, there will be a new button, called "Step". This will execute the current file in the single-stepper, allowing you to easily experiment with the stepper's operation.

If VF hasn't been opened since the last reset, the Editor may not be able to detect the presence of the stepper, and the Step button won't appear. If this happens, close the Editor, open VF, then open the Editor again.

-=[VF+INTERCEPT 1.2ß1]=-
• The big change here is, of course, the single-step debugger. See the separate Stepper documentation for details on how to use it: the documentation here will focus on the commands to invoke it.

• The existing Add Intercept command now has a "Single-step" option in its popup of intercept types that can be installed: this is one of the three ways in which the stepper can be invoked. This option works just the same as other intercept types, including the use of conditional expressions, and is covered by the existing VF+Intercept documentation.

To create a single-step intercept by directly calling the `VF:Intercept()` method, use the symbol `'step`.

• New command: Step Into Next Button
Available: always (it really should check for compatibility before appearing, but it has to be always available in order to be usable from the current BugDrop's Additions menu).
Action: the next time any button is pressed (with the pen only - keyboard equivalents for buttons currently cause a hang), it will pop up a list of methods that might be called (directly or indirectly) by the button. Selecting a method will open an instance of the single-stepper for that method.

For the purposes of this command, a button is defined as a view that calls `:TrackHilite()` or `:TrackButton()`, typically from its `viewClickScript`: the command works by installing root-level intercepts for those methods. Additional methods may be intercepted in future versions in order to detect other tappable items, such as Extras Drawer icons. Buttons that are a part of VF itself are ignored, so that you can safely close VF after giving this command.

The methods listed in the popup will include the view's `buttonPressedScript`, `buttonClickScript`, `pickActionScript`, and `pickCancelledScript`, if defined. It will also include any methods in the same view that might be called by these methods, or by the view's `viewClickScript` (determined by looking for symbols in the function's literals array that name other functions in the view). The process is applied recursively to any called methods, and the methods they call, and so on. Functions with no more than 3 bytes of instructions (in other words, that do nothing more complicated than "return NIL") are not included in the list: do-nothing functions like this are commonly found as placeholders in system protos, but there's no point in stepping into them.

It is possible to select a method to step into that is never actually called. For example, you could select the `pickActionScript` of a popup button, but cancel the resulting popup menu so that only its `pickCancelledScript` gets called. In cases like this, the stepper instance created for the uncalled method will remain on the screen, with most of its displays being blank and most of its buttons doing nothing. You'll need to close this view in order to return to normal operation.

To invoke this command from another debugging tool, such as the key equivalent for it in VF+Keys, use the following code:

```
     local step := VF:MapAdditions('GetObject, '[StepNextButton], true);
     if step then call step with ();
```

• The final way of invoking the single-stepper is under direct program control. VF+Intercept installs the following method in VF's base view:

```
     VF:MakeStepper(function, name)
```

function - The function to be stepped into. Note that this must be an actual reference to a function, not the name of a method. Also note that you are still responsible for actually calling the function: the stepper window will be open but unusable until this happens.
name - A string describing the function: this will be used as the title of the stepper window.

Note that you may need to have opened VF at least once before this method is available.

Example of use: Let's suppose you're having trouble with the `viewSetupChildrenScript` of a view, and want to single-step through it every time your program runs, rather than having to install an intercept for it every time you load a new version of the program. You could place the following code in some script that executes earlier, such as the `viewSetupFormScript` of the same view:

```
     GetRoot().|ViewFrame:JRH|:?MakeStepper(self.viewSetupChildrenScript,
          "MyView:vSCS");
```

• The Install Intercept command has a new option: you can assign a sound to be played every time the intercept is triggered. Under Newton 2.x, the sound can be chosen from the list of registered system sounds. Under Newton 1.x, the sound can be chosen from a built-in list of sounds, but this isn't working at the moment. The sound is played synchronously, so that it is always heard in its entirety even if multiple intercepts are triggered in a row.

• The final parameter to the `VF:Intercept()` method, for adding intercepts under program control, has been redefined to allow specification of sounds and other future options. This parameter was previously defined as being a string containing a conditional expression, or NIL. These values are still accepted, but it can also be a frame containing the following slots:

- `condition` - A string giving a conditional expression: if NIL, missing, or empty, the intercept will be unconditional.
- `sound` - NIL for no sound, or an integer or symbol specifying a sound to play. An integer specifies the magic pointer number of a system sound: this option is intended for Newton 1.x, but works under any system version. A symbol is passed to `GetRegisteredSound()`, a Newton 2.x-specific function, to get the actual sound to play. To see what the valid symbols are, execute `SoundList()` on your Newton and examine the resulting array. Unrecognized symbols will produce the standard system beep. If the Newton text-to-speech software is ever released, this slot will also accept strings to be spoken when an intercept is triggered.

• There is an error in the existing VF+Intercept documentation, concerning which methods can and cannot be intercepted. It claims that you cannot safely install an intercept in a place where it will be found via _parent inheritance. This was based on an incorrect understanding of what value `self` has in an inherited: method call: actually, such intercepts work just fine. You can even globally intercept methods in the root view, but be careful. It would not be a good idea to install a single-step intercept in the root view, since if the stepper itself called that method an infinite recursion would result.