-=[VF+FUNCTION 1.2ß2]=-

• This update simply makes a few tweaks to the function lister so that it's more accurate. My test routine (which decompiles and recompiles every built-in NewtonScript function) has been tried on a MP110, 120, and 2000, with only one failure each, all of which seem to be outside my control.

• The previous version attempted to display small binary contants (such as patterns) inline using the `MakeBinaryFromHex()` compile-time function, rather than as external objects. This didn't work, and I later realized that it made my test routine unusable (since the function doesn't exist in the Newton), so I took this feature out.

• There is no longer an apparent assignment of a variable to itself at the start of functions containing a closure that accesses one of the function's parameters. Actually, this is a copy of the parameter to a local variable of the same name, since 2.x closures can't directly access their parent's parameters.

• String concatenations which are unnecessarily done in multiple parts, such as:

```
A & (B & C & D) & E
```

where the inner three items are first concatenated together and then concatenated with the outer two items, are now properly parenthesized to indicate this. Previously, the code above would have appeared as just `A & B & C & D & E`. Note that current versions of the NewtonScript compiler, such as `Compile()` in the MP2000, are smart enough to know that parentheses like this are unneccessary, and will ignore them. This accounts for my one test failure on the MP2000: its ROM was built with a version of the compiler old enough to not have this optimization, so the one function containing a split-up concatenation can't be recompiled into anything like its original form.

-=[VIEWFRAME 1.3ß3]=-

• Dragging to select a view now works from VF's minimized state and the ViewFinder, in both cases by dragging from the Maximize button. In ViewFinder, if a view is selected (either directly or returned by a `|GetValue:ViewFrame:JRH|` method in that view), it will be turned over to ViewFinder's selection mechanism, and the view hierarchy can be browsed starting at that point. Otherwise, the selected object is displayed in VF.

• Drag-selecting a view now ignores VF itself. This means that you can now use this feature (although perhaps not very accurately) even when VF is zoomed to fill the screen: it will select whatever is underneath VF. Hopefully you'll be able to tell from the size and position of the selection box exactly what is being selected.

• When VF's scroll arrows are hidden when the end of the display has been reached, they are now explicitly unhilited before hiding them. This fixes a hiliting problem when scrolling via VF+Keys' keyboard equivalents: `PressButton()` doesn't unhilite the button if it hid itself, so the arrows were being left in an odd state (hilited but hidden).

• Fixed a bug with displaying an object via a direct VF call prior to the first time VF has been opened since the last reset.

• New display list item: checkboxes. This is used extensively in the new Set/Clear Breakpoints command in VF+Intercept. To add these from your own `DisplayObject` or `AnnotateObject` scripts, use:

```
VF:AddCheck(text, value, state)
```

text - the label for the checkbox.
value - an identifying value for this checkbox. It should be a symbol based on your registered developer signature, or a frame or array with such a symbol as its class.

state - specifies the initial state of the checkbox: NIL = unchecked, non-NIL = checked.

When a checkbox changes state, the following message is broadcast to all Addition frames:

```
CheckHit(value, state, obj, VF)
```

- *value*: the *value* parameter as originally passed to `AddCheck`.
- *state*: the current state of the checkbox.
- *obj*: VF's current object.
- *VF*: VF's base view.
- Return value: non-NIL to indicate that you've handled the message, otherwise NIL to allow it to be passed to other `CheckHit` methods.

If no `CheckHit` method handles the specified *value*, there is one built-in handler: if *value* is an array, its class is used as the name of a global function that is called, using the elements of the array as parameters. Two specific character constants have special meanings in the array: `$%` is replaced with VF's current object, `$$` is replaced with the checkbox's current state. Example: to make a checkbox that shows and sets the `BreakOnThrows` global variable, use:

```
VF:AddCheck("BreakOnThrows",   // label
     [DefGlobalVar: 'BreakOnThrows, $$], // calls
DefGlobalVar('BreakOnThrows, state)
     GetGlobalVar('BreakOnThrows)   // initial value
);
```

-=[VF+INTERCEPT 1.2ß2]=-

• Some major internal changes have been made to the stepper. Stepper windows are now treated as a pool, with the appropriate one brought to the front for each breakpoint, instead of acting as a pure stack. Also, breakpoints from ALL open steppers are set, rather than only those from the stepper that thinks it's for the currently executing function. This change means that stepping will get slower the deeper you step in (but I haven't really noticed a problem with this). However, it also means that unexpected jumps, such as an exception caught by a different function than threw it, no longer result in all steppers being disconnected. I would have had this update ready for you a week ago, if it wasn't for some extremely obscure errors (especially in the Close In command) resulting from these changes. Please note that things that worked before might be broken now: let me know of any problems you find, and any suggestions you have for making the stepper useful in your development work.

• On Newtons with a soft button bar (such as the MP2000), stepper windows now cover the button bar (which can't be used while the stepper is open, anyway).

• New key equivalent: 'V' pops up the View: menu, allowing you to select a different view for the main list window. Note that there is currently a problem here, and with all other popups in the stepper: it loses the current key view, so the next action has to be done with the pen. This is especially annoying with the caret popup in the value editing slip, since the selected character ends up going nowhere! This seems to be a general problem with popups in modal dialogs: I'll keep looking for a workaround.

• The Call Trace listing is now available in the main stepper window. This shows you information about the function being stepped, then the function that called it, then the function that called that one, and so on back to some function called in response to a system event. Six items are shown for each function: tapping on any of these displays them in the Work Area. Holding the pen down on them does nothing: the only call trace item which is physically possible to modify is the program counter value, and that is much more safely modified by the new Skip To feature, described later. The items displayed are:

- The name of the function. This will be "CURRENT FUNC" for the function displayed in the stepper, which appears at level 0. Functions for which a stepper window currently exists will be shown as "stepper = " plus the title of the window. Other functions will have various techniques tried in order to assign a meaningful name to them. Tapping on this item will eventually pop up a list of call level-related options, such as stepping out to a given function.
- A reference to the function itself.
- An array of the function's parameters. Modifying this array will NOT have any effect on the function, however modifying individual elements of the array might. TODO: if the name of the function's parameters can be determined, use a frame here instead of an array.
- The program counter (PC) value of the function. This will always be -1 for native functions.
- The receiver (self) of the function, or NIL if this was a function call rather than a message send.
- The implementor (frame containing the function), again only if this was a message send.

• Editing of locals and stack values now works! Just hold down the pen on any item in those two windows, and you'll get a value editing slip. This lets you change a condition between TRUE and NIL, change the limit of a FOR loop so it doesn't run as long, change the value about to be returned from the function, etc.

• The keyboard button in the editing slip has been removed, as it didn't work (apparently the only way to get a keyboard open in front of a modal dialog is to make the keyboard modal, which has the side-effect of losing the current key view so the keyboard becomes useless). If no hardware keyboard is connected, the slip will have an embedded keyboard. Note that there are still some editing glitches here, apparently fundamental problems with editing inside a modal dialog: the caret popup is useless, and the correction popup won't appear at all.

• The editing slip now has buttons labelled £, Å, and §, which will pop up lists of the function's literals, arguments, and stack values, respectively. Selecting any item will insert an expression referencing that item into the entry line. Note that the reference should only be used for its value: assigning to one of these references won't have any effect.

• Some changes in the popup you get when you hold down the pen on a line in the Trace History or Function Listing:
- "Run until step *N*" changed to "Trace until step *N*". "Run until" would better describe a planned feature that would run the function at nearly full speed, but without updating the trace history.
- Trace Until runs considerably faster than Run Until used to. It was updating the locals & stack listings after each step for no good reason.
- A new Skip feature that allows you to jump around within a function. This can be used to make an early exit from a function, go back and retry a function call with different parameters, etc. In order to avoid crashes, this command requires that skips only be made to other points in the function at which the stack depth is the same. I hope to relax this requirement in the future (by pushing or popping the stack as necessary to get the proper depth), but I'm not going to make any promises right now as the code to do this would be VERY nasty. The possible ways this command can appear are:
-- "Skip to step *N*", if the current stack depth matches the calculated depth for the selected step.
-- If the current stack depth is 1 too low to permit a skip, but the selected step starts with a pop (shown as a semicolon), a skip will be allowed to the instruction just after the pop. This is shown as "Skip to step *N*+1".
-- "Can't skip (pop *N* required)" or "Can't skip (push *N* required)" indicate that a skip can't be done to the selected step, and the amount of stack imbalance is shown. You may be able to successfully skip to a nearby instruction, or by taking another step or two before skipping.
-- "Can't skip (step is unreachable)" indicates that you're trying to step to an instruction that cannot

be reached in normal execution of the function. I believe that the only situation where this can occur is immediately after a break statement. There's nothing really preventing a skip to such an instruction, the problem is that the stepper has no way of calculating what the stack depth is supposed to be at that point.

• In order to support the Skip To command mentioned above, the stepper now calculates what the stack depth should be for each instruction. You can see this value displayed by tapping on any line in the Trace History or Function Listing. I think this is working properly, however it's quite possible that there are some NewtonScript constructs for which the calculation fails. If you hear any SysBeeps, either while a stepper window is opening or while stepping, this indicates that a stack depth inconsistency has been detected. If this happens, please duplicate the condition with an Inspector connection if needed, and send me the Inspector output (and the source code to the function being stepped, if practical).

• There is preliminary support for meaningfully stepping into system routines that make function calls of their own. For example, stepping into a `Perform()` call will actually step into the method being performed, rather than trying to step into the `Perform` global function itself (which won't work since it's a native function). There are still a lot of problems in this area: in particular, with any functions that call another in a deferred manner (Deferred, Delayed, and Procrastinated actions, etc.), you can only use Close In: Step In will leave the Newton in a weird state where all open views will continue to work, but you can't open any new views. Some other gotchas with this feature:
- Some of the array-related global functions can have two parameters that can be a function: *test* and *key*. If both are functions, only *test* will be stepped into (this may be fixed).
- The function will only be stepped into once, even if it's called multiple times.
- It's possible that the function will never be called at all, such as an array test applied to an empty array. In this case, a stepper will be created but never opened. This isn't really a problem, but it wastes memory (until 5 seconds after the last stepper window is closed, when a cleanup routine runs).
- Any kind of deferred or delayed function call will actually occur before the next step is taken, whether you're stepping into it or not. This is much earlier than the call would normally occur, and it's possible that the call will fail due to some information it needs not being set up yet. It may eventually be possible to step into such a call, then switch back to the calling function and complete it before stepping further into the deferred call.
- Some of the functions require an additional action to actually trigger the call: power on/off functions, and undo actions (not implemented yet; will require hitting Cmd-Z on a keyboard). There are two windows of opportunity during which you can perform the trigger action and successfully step into the function: before the next step is taken (but note that only Close In will currently work right), and within 5 seconds of closing the last stepper window. At any other time, the triggered action will run at full speed.
- I've tested only a few of these steppable functions so far. Please let me know of any successes or failures you have with the use of this feature.

Here are the currently implemented global functions which can be stepped into:
- Apply, [Proto]Perform[IfDefined].
- AddDeferred(Action|Call|Send) - NOTE: Use Close In only for now.
- Add(Delayed|Procrastinated)(Action|Call|Send) - Close In only, delay time set to 1 for your convenience.
- AddPowerOffHandler, RegPower(Off/On) - Requires power switch hit to actually trigger the call.
- ArrayPos, all array functions starting with 'B' or 'L', all variations on Sort - Steps into first call to a *test* or *key* parameter defined as a function.
- AsyncConfirm, DoProgress, Map, MapCursor.

- DoPopup - steps into pickActionScript.

Some view methods can also be stepped into:
- Open, FilterDialog, Modal Dialog - steps into first of viewSetupFormScript, viewSetupChildrenScript, or viewSetupDoneScript that is actually defined in the view (_proto inheritance only). Eventually it may step into all of the three scripts that are defined. TODO: provide some indication of which script was actually stepped into.
- Close, Show, Hide, Hilite - steps into corresponding script.
- SyncChildren -> viewSetupChildrenScript.
- SyncView -> viewSetupFormScript.
- PopupMenu -> pickActionScript.
- SetupIdle -> viewIdleScript, with delay set to 1 so you don't have to wait.
- DoDrawing, Effect, RevealEffect - steps into specified drawing method, I suspect these may not work due to view clipping.

I know I've missed several suitable functions and methods (undo actions, RedoChildren, SetValue, ScrollEffect, etc.): I'll work on adding those, and testing all of the existing ones, as soon as I get the basic step-in functionality working a bit more reliably.

• New Addition command: Set/Clear Breakpoints
Available: Always (if NS Debug Tools is installed), but more features are available if the current object is an interpreted function, or a frame (such as a view or template) containing such functions.
Action: Displays options for setting the Newton's debugging state, and setting or clearing breakpoints. The display will contain the following items:
- Static text: "Breaks handled by", followed by the name of the package (or "ROM", or "unknown") that owns the current BreakLoop() global function.
- Possibly a list of buttons for setting other breakpoint handlers. This list will include VF+Intercept, and any other packages which have overridden the BreakLoop() global function. Note that there is no immediate feedback from using one of these buttons: you'll need to redisplay the options (Fmt->Normal) to see the change acknowledged.
- Checkbox: "Breakpoints enabled", which shows and sets the breakpoint state as controlled by GloballyEnableBreakpoints(). Its effect is immediate.
- Checkbox: "BreakOnThrows", which shows and sets this global variable. If you set this and the preceeding checkbox, and set VF+Intercept as the break handler, you'll be in a "StepOnThrows" mode where you can see exactly what function threw an exception, and the stack trace leading up to it, without an Inspector connection.
- Checkbox: "Set 1-time breakpoints". If checked, any breakpoints set by the following checkboxes will trigger only once and then delete themselves. This is on by default, so that you don't get into an infinite loop of a breakpoint being triggered by the handling of that breakpoint. Changing the setting of this option does not affect any existing breakpoints. BUG: this checkbox appears even if there aren't any breakpoints that can currently be set.
- Checkboxes for breakpoints that can be set. If the current object is a bytecode function, there will be a checkbox for each function step as would be displayed in the single-stepper (but note that you could still use NS Debug Tools to handle the breakpoint). If the current object is a frame, there will be one checkbox for each bytecode function within the frame (including those found by _proto inheritance): checking it will set a breakpoint at the entry point of the function, unchecking it will clear all breakpoints in the function. If the current object isn't a function or frame, or doesn't contain any suitable functions, no checkboxes will appear here.
- If any breakpoints are currently set, the display will end with buttons for clearing them. "Clear these" will appear if any of the breakpoint checkboxes above are initially checked. "Clear ALL" will appear if any breakpoints are set anywhere, and will include the current number of set breakpoints

in parentheses.

Note that breakpoints you set will take effect instantly, if breakpoints are currently enabled. You can immediately perform some action that results in a call to the specified function, and it will be single-stepped or displayed in the Inspector depending on your break handler setting. You don't have to back up or do anything further in VF, although if you're using the stepper it would be a good idea to close VF before triggering the breakpoint so that the maximum amount of memory is available.

It's fairly easy for the breakpoint display to get out of sync with the current breakpoints, especially if 1-time breakpoints are being used. Remember that you can refresh the display by tapping Fmt then Normal.

• If you manage to invoke a single-stepper on a native function, such as via an exception thrown due to bad parameters, it will no longer die horribly. The only stepper displays that will work in this case are the parameters and the call trace (which will automatically be displayed instead of the trace history). This will let you see what went wrong with the call, and how you got there. Tapping Step will close the stepper and continue with the next most recent stepper window, if any. It should soon be possible to step out to a specific function (presumably one containing an onexception handler that will catch the exception), even if that function hadn't already been stepped into.